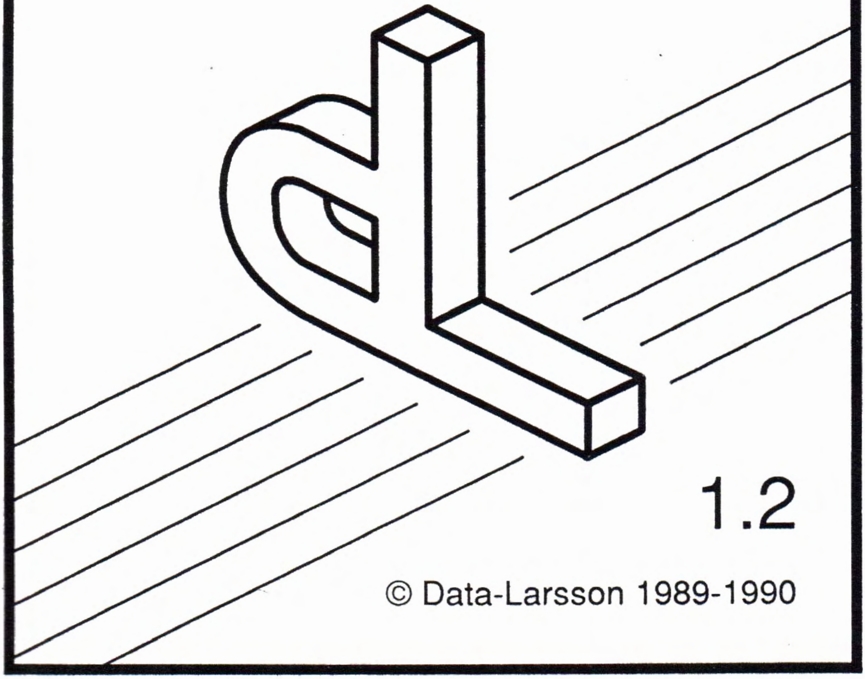
DL-Pascal

for the

PB-2000

User's guide
Reference manual



1.2

© Data-Larsson 1989-1990

DL-Pascal

PB-2000

User's guide

Reference manual

Preface

The manual and contents of the ROM card of the DL-Pascal compiler are protected under copyright laws. Copying of this product is strictly forbidden.

Data-Larsson and Casio Computer Co., Ltd. warrants that this ROM card is an accurate copy of the original. No other express or implied warranties are given, including performance, accuracy, functionality, or fitness for any particular purpose.

Data-Larsson and Casio Computer Co., Ltd. assumes no responsibility for any damage or loss resulting from using this product, such as consequential, incidental, exemplary, and special damages, including lost profits. This also includes loss or damage caused by deleted data as a result of malfunction, program errors (both system and user), repairs, and battery replacement. For best protection, be sure to back up important programs and data.

The author of the DL-Pascal compiler and manual is Hans Larsson, Sweden.

Trademarks

The following trademarks have been used in this manual.

IBM is a registered trademark of International Business Machines Corp.
Turbo Pascal is a registrered trademark of Borland International.
QuickPascal is a registered trademark of Microsoft Corporation.
EPSON is a registered trademark of the Seiko Epson Corporation.
CASIO is a registered trademark of Casio Computer Co., Ltd.
DL-Pascal is a trademark of Data-Larsson.

Copyright © 1988-1990 by Data-Larsson, Sweden. All rights reserved.

Table of Contents

Part 1
Using DL-Pascal

1. Introduction 3
Pascal 4
Notation 5
Manual structure 6

2. Computer and ROM Card 7
PB-2000 Key Layout 7
PB-2000 Bottom view 11
Precautions 12
Starting DL-Pascal 12

3. Programming in DL-Pascal 15
The First Program 16
The Second Program 19

4. Separate Compilation 27
Creating the Unit 28
Writing the Program 30
Writing Another Program 31

Part 2
Operating System

5. Introduction 35
Programmable Keys 35
Input Line Edit 36
File Specifier Structure 37
Printer File Name 39
RS-232C File Name 39

6. MENU Mode 43
Selecting a file 43
Using the EXE key 45
[edit] Edit Function 45
[new ed] New edit Function 45
[compil] Compile Function 46

[disk] Disk Function 46
[name] Rename Function 46
[kill] File Delete Function 47
[load] Load Function 47
[save] Save Function 48
[append] Append Function 48
[cnv fr] Convert Function 49
[cnv to] Convert Function 49
[print] Print Function 50
[prog.k] Programmable Key Function 50
[memory] Memory Function 51
[set] Data Set Function 52
[preset] Preset Function 53
Disk Menu 53
[compil] Disk File Compile Function 54
[name] Disk File Rename Function 54
[kill] Disk File Delete Function 54
[load] Disk File Load Function 54

7. CAL Mode 55
Cursor movement 55
Data entry 56
Normal calculations 57
Variables 58
Mathematical functions 59
Commands 61
ANGLE 62
APPEND 62
BEEP 62
CLEAR 62
CLS 63
COMP 63
CONVFR, CONVTO 64
COPY 64
DEBUG 65
DEFKEY 65
DEL 66
DO 66
DIR 66
EDIT 67
FORMAT 67
PRESET 67
REN 68
RUN 68
VERIFY 69

8. Batch Files **71**
- ECHO, ECHOX 72
- GOTO 73
- IF 73
- IFEXIST 73
- PAUSE 74

9. Editor **75**
- Cursor movement 76
- Data entry 77
- [search] Search Function 77
- [block] Block Function 80
- [compil] Compile Function 84
- [insert] Insert File Function 84
- [goto] Goto Function 84
- [setup] Setup Function 86

10. MEMO Mode **89**
- Input to MEMO file 89
- Searching in MEMO file 90

11. Function Memory **93**
- Calculate Using Function Memory 93
- Function Memory and MEMO mode 95

12. Compiler **97**

13. Debugger **99**
- [step] Step Function 100
- [data] Data Window Function 104
- [raw] Raw Window Function 105
- [search] Search Function 107
- [goto] Goto Function 107
- [setup] Setup Function 108

14. Expanding the PB-2000 . . . . **109**
- ROM Cards OM-51P, OM-53B 109
- RAM Pack RP-33 109
- Interface Units FA-7, MD-100 110
- Serial interface 112
- Parallel interface 113
- Cassette Interface (FA-7) 114
- Disk Interface (MD-100) 114

Part 3
Compiler Reference

15. Introduction **119**
- EBNF 119

16. Symbols . . . 121
Identifier . . . 121
Number . . . 123
String . . . 124
Single-character symbol . . . 124
Double-character symbol . . . 125
Comment . . . 125

17. Program . . . 127
Program heading . . . 127
Uses declaration . . . 128
Declarations . . . 128
Body . . . 129

18. Label Declaration . . . 131

19. Constant Declaration . . . 133

20. Type Declaration . . . 135
Ordinal Type . . . 136
Integer types . . . 136
Character type . . . 136
Enumerated type . . . 137
Subrange type . . . 137
Real Type . . . 138
Pointer Type . . . 138
String Type . . . 139
Procedure and Function Type . . . 140
Record Type . . . 140
Array Type . . . 142
Set Type . . . 143
File Type . . . 143
Type Compatibility . . . 144

21. Variable Declaration . . . 147
Absolute Variables . . . 147
Initialized Variables . . . 148

22. Procedures and Functions . . . 151
Parameters . . . 152
Value Parameters . . . 152
Variable Parameters . . . 153
Untyped Parameters . . . 153
Open Array Parameters . . . 154
Subroutine Body . . . 155
Forward-declared Subroutines . . . 156

23. Expressions 159
Constant Expressions 160
Binary Operators 160
Unary Operators and Arguments 166
Variable Reference 169
Variable Dereference 170

24. Statements 175
Compound Statement 175
If Statement 176
Case Statement 177
Repeat Statement 178
While Statement 178
For Statement 179
With Statement 180
Goto Statement 181
Inline Statement 182
Assignment Statement 183
Procedure Call 183

25. Interface and Implementation Units 185
Interface Unit 185
Implementation Unit 186
Using Units 187
Circular Import 187
System and Crt Units 188

26. Dynamic Memory Allocation 189
New and GetMem 190
Mark and Release 190
Dispose and FreeMem 191
MaxAvail and MemAvail 192
HeapError 193

27. Files 195
IOResult Error Handling 195
Typed Files 196
Untyped Files 197
Text Files 198
User Text Type 200

28. Program Termination 207
Halt 207
RunError 207
ExitProc Variable 208

29. Windows 209
Setting a Window 209
Moving in the Window 210

30. Graphics 211
Plot Pixels 211
Read Pixels 212
Draw Lines 212
Direct Graphic Access 213

31. System Unit 215
System Types 215
System Variables 216
angle 216
errorAddr 216
exitCode 216
exitProc 216
heapError 217
input 217
output 217
System Procedures and Functions 218
Abs 218
Addr 218
Append 218
ArcCos 219
ArcCosH 219
ArcSin 219
ArcSinH 219
ArcTan 220
ArcTanH 220
Assign 220
BlockRead 221
BlockWrite 222
Ceil 222
Chr 222
Close 223
Concat 223
Copy 224
Cos 224
CosH 224
Dec 224
Delete 225
Dispose 225
Eof 226
Eoln 226
Erase 226
Exit 227
Exp 227
FilePos 228
FileSize 228
FillChar 228
Floor 229

Frac . . . 229
FreeMem . . . 229
GetMem . . . 229
Halt . . . 230
Hi . . . 230
Inc . . . 231
Insert . . . 231
Int . . . 232
IOResult . . . 232
Length . . . 232
Ln . . . 232
Lo . . . 233
Log . . . 233
Mark . . . 233
Max . . . 233
MaxAvail . . . 234
MemAvail . . . 234
Min . . . 234
Move . . . 235
New . . . 235
Odd . . . 236
Ofs . . . 236
Ord . . . 236
Page . . . 237
ParamCount . . . 237
ParamStr . . . 238
Pi . . . 238
Pos . . . 238
Power . . . 239
PowerInt . . . 239
Pred . . . 239
Ptr . . . 240
Random . . . 240
Randomize . . . 241
Read (typed file) . . . 241
Read, ReadLn (text file) . . . 241
RealError . . . 242
Release . . . 242
Rename . . . 243
Reset . . . 243
Rewrite . . . 244
Round . . . 244
RoundTo . . . 244
RoundDigit . . . 245
RunError . . . 245
Seek . . . 246
SeekEof . . . 246

SeekEoln 246
Seg 247
SetTextBuf 247
Sin 247
SinH 248
SizeOf 248
Sqr 248
Sqrt 249
Str 249
Succ 250
Swap 250
Tan 251
TanH 251
Trunc 251
UpCase 251
Val 252
Write (typed file) 252
Write, WriteLn (text file) 253

32. Crt Unit 255

Crt Variables 255
dispStatus 255
keyStatus 256
textAttr 257
Crt Procedures and Functions 257
AssignCrt 257
ClrEol 258
ClrScr 258
Cursor 259
DelLine 259
Display 260
GotoXY 260
InsLine 260
KeyPressed 260
Line 261
NextKey 261
Plot 262
Point 262
ReadKey 262
Sound 263
TextDisplay 263
TimerOut 263
TimerStart 264
UpdateRegion 264
WhereX 264
WhereY 265
WindMaxX 265
WindMaxY 265

WindMinX 265
WindMinY 265
Window 266

33. Compiler Options 267
Compiler flags 267
Conditional compilation 269
Compiler directives 271

34. Inside PB-2000 DL-Pascal 273
Data Formats 273
Parameter Formats 276
Register Saving 277

35. Comparing to Other Pascals 279
Standard Pascal 279
Turbo Pascal and QuickPascal 280

Part 4

Appendix

A. Batteries 285
Checking the Power 285
Changing Batteries 285
Using an AC adapter 287

B. ROM Cards 289
Replacing the ROM card 289
Starting the ROM card 290

C. Key Codes 291
Normal Keys 291
Programmable Keys 294
Cursor Keys 294
Other Keys 294
Power-On Key Code 295

D. Characters and Attributes 297
Character sets 297
Attributes 299
Character Display Buffer 299
Programmable Characters 300

E. Error Messages 303
Run-time Error Codes 303
Compile Error Codes 309

F. System Memory Area **317**
- Heap Memory Area 317
- Stack Memory Area 317
- Numeric Variable Area 318
- System Memory 318

G. CPU Instruction Codes **321**

Index **325**

Part 1

Using DL-Pascal

Chapter 1

Introduction

Welcome to the DL-Pascal compiler for the PB-2000! DL-Pascal has been developed to take advantage of a compiling language on the PB-2000. A compiling language has a lot of possibilities over an interpreting language (such as BASIC):

- Building of library units, that can be easily used from different programs.
- Programs running up to 500 times faster than an interpreter program.
- No need to reveal the source code, as the program is converted to machine code before it is run.
- Type checking of variables and values. More errors can be found before the program is being run as variables and values must be consistently used in respect to their type.
- Structured programming makes it easier to read and maintain the program.
- More operators and types, to make program writing easier.

The programming languages on pocket computers have earlier been interpreting languages, such as BASIC. The difference between an interpreting and a compiling language is the way they evaluate the program being run. With an *interpreting* language, each line in the program is decoded when the program is run. This means that the computer is looking at what the line is supposed to do, checks the syntax of the line, and then executes the line. This has to be made for each line being executed, whether it has already been decoded earlier or not. As a result, an interpreting language executes a program rather slowly. It is like a person reading a book in a foreign language, and having to look up every word in a dictionary.

A *compiling* language, on the other hand, converts the program after it has been written, and before it is actually being run. The resulting code can then be generated in such a way the computer can understand it much easier than normal text, and therefore the program can run much faster. The syntax of the language can also be checked at compilation time.

The main disadvantage of a compiling language is the compilation process, which can take some time, especially for a large program. It is not so much of a problem, as this translation is only needed once. After compilation, the program can be run any number of times without any need for compilation (at least not unless any program errors must be corrected). This is as if the person has translated the foreign book to his own language, and therefore is able to read it fluently.

Pascal

The Pascal programming language was developed by Niklaus Wirth around 1970. The language was developed to teach structured programming. Most elements of the language was brought in from other programming languages, such as ALGOL.

Pascal was from the beginning designed to be a compiling language. During compilation, the program can be thoroughly checked for consistent use of variables and values (called *strict type checking*). Pascal takes advantage of this to catch as many errors as possible before the program is actually run. The strict type checking in combination with the structured programming style makes it easy to develop and maintain programs, and soon Pascal became the main programming language for educational purpose.

The success for the Pascal language has resulted in a number of implementations of Pascal on different computers, and also in some new programming languages following the style of Pascal. Among such languages are Wirth's new programming language *MODULA-2*, and the language *ADA* supported by the United States' Department of Defense.

As implementations of the Pascal programming language have been available for such a long time, Pascal is still the most wide-spread structured language. It is also the language used in schools in many countries.

Turbo Pascal

One of the most successful implementations of the Pascal language is Borland's *Turbo Pascal.* This was the first compiling language to successfully integrate a powerful editor with a very fast compiler and a very good price. In addition, it had a number of enhancements to the language to make it easier to develop large programs.

Since the first version of the Turbo Pascal for the CP/M and MS-DOS/PC-DOS operating systems, more than one million copies of Turbo Pascal have been sold. This has had the result of Turbo Pascal being the de-facto standard Pascal on these computers.

DL-Pascal

This implementation of Pascal has a number of enhancement to the standard Pascal language. Due to the restricted memory available in the PB-2000, however, a few differences exist. These differences are only on small details, as can be seen in chapter 35. Instead, more useful features have been included, some of them are also included in Turbo Pascal 5.5 and QuickPascal 1.0. Some of these features are:

- Separate compilation of library units
- Powerful string handling procedures and functions
- Free order of declarations
- Initialized variables
- Procedure and function variables
- Expressions allowed in constants
- Type conversion of expressions and variables
- Direct access to system memory
- Inline machine code

In addition, the DL-Pascal ROM card contains a full-screen editor, file menu system, large calculation area, batch file processing, function calculation memory, memo note pad with search, and a symbolic program debugger. All this is available in this single ROM card!

Notation

In this manual, different type faces are used to show the type of text:

Boldface text	Used to emphasize names in the text: The **MENU** mode is used to navigate through the files of the computer.
Italic text	This is used to introduce new terms in the text: DL-Pascal is a *compiling* language.
Sans-serif font	Used for Pascal reserved words and identifiers within text: The System unit is predeclared.
Sans-serif italic	Text to be entered at the keyboard is shown in this way: Enter *PAS*.
`Fixed pitch`	Program listings and function keys are displayed using this font: Press `[compil]` to start compilation.

In addition, the following characters are used to specify different things in the text and program.

CAL	Special keys on the computer: Press CAL to start the **CAL** mode.

In this manual, the key S always means the red shift key (located to the right of the CALC key). The letter key *S* is always written in sans-serif italic, as specified above. The keys of the computer are described in detail in the next chapter.

...

Ellipses are used in program listings to mark that other program lines can be added, but are not shown in this listing:

```
BEGIN
   ...
END
```

$number

Specifies a hexadecimal number instead of a decimal number:

The number $1A is equal to 26.

#number

This specifies a character having the entered ordinal number (as can be seen in appendix D):

Character #$41 is the uppercase A.

Manual structure

This manual has four parts. The first part is the one you are reading now. It is an introduction to the PB-2000 computer and the DL-Pascal compiler. Some program examples are introduced in the next chapters. You can work through these examples to learn how to use the file menu system, the editor, the compiler, and also the debugger.

The second part describes the operating system used in the DL-Pascal ROM card. Within the operating system, you have the file menu mode, calculation mode, batch file processing, note pad functions, editor, compiler starting, and debugger.

Next part describes the DL-Pascal language in detail. Use this part as a reference to DL-Pascal.

Last part is the appendix. In this part general functions are described, such as: battery and ROM card change, key codes, character set and attributes, error messages, and system memory structure. The appendix can also be used as a reference.

Chapter 2

Computer and ROM Card

In addition to this manual, this package should include either:

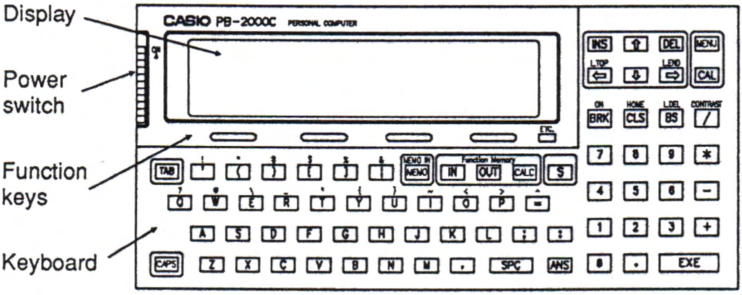
- PB-2000 with built-in DL-Pascal ROM

or

- DL-Pascal ROM Card (PB-2000 computer will also be needed)

If the batteries has not yet been inserted into the PB-2000, do this now. This is explained in appendix A.

PB-2000 Key Layout


Display
Power switch
Function keys
Keyboard
CASIO PB-2000C PERSONAL COMPUTER
INS
DEL
MENU
CAL
BRK
CLS
BS
TAB
MEMO
IN
OUT
CALC
S
CAPS
SPC
ANS
EXE

The *power switch* is slided upwards to switch power on, and downwards to switch power off. The computer polls the power switch state, so sometimes when the PB-2000 is running a program, it can take some time before power is actually switches off. Power is automatically switched off after som time of inactivity. The ON BRK key is then used to switch power on again. The power-off time is set in the MENU mode, as described on page 52.

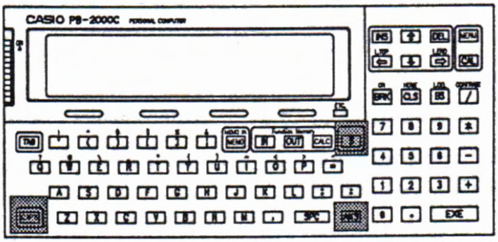
The PB-2000 can display 4 lines of 32 characters. Graphics can also be displayed, with 192 pixels horizontally by 32 pixels vertically. The current position in the display is shown by a flashing or solid *cursor*. It depends on the mode and input states how the cursor is displayed.

Below the display are four unmarked *function keys*, and the key ETC. These keys are used to select one choice displayed on the bottom line of the display (the *function menu*). These choices are displayed within square brackets. Some choices can be *dimmed*, which means that the dimmed key cannot be used at that moment.

The keys on the *keyboard* can be pressed at any time, even during execution of a command or program. This is the *type-ahead* function of the PB-2000, and makes the pocket computer work as a large computer. A keyboard buffer of up to 64 key codes stores the pressed keys.

The keyboard keys can be grouped by function. The placement of the keys in respect to the groups are:

Shift keys



S *Shift*. This key is used to shift the function of the next key pressed. The flashing cursor shrinks to signal that the shift key has been pressed. Another press on this key cancels the shift state. For most keys the shifted function is written in red above it. When used with the lower two rows of alphabetic keys (*A* to colon and *Z* to SPC), user-programmed key sequences are returned.

In this manual, the symbol S always means the shift key. The letter key S is always shown as *S*.

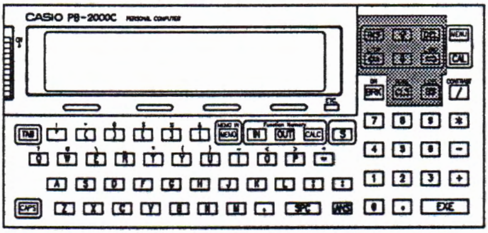
ANS *Alternate shift*. This key is used to get an alternate function of the next key. The flashing cursor shrinks to signal that the alternate shift key has been pressed, and another press on this key cancels this shift state. The alternate function of the keys are not shown on the keyboard, but are described in appendix C.

CAPS *Case shift toggle*. A press on this key toggles the alphabetic keys between lower and upper case. The flashing cursor will have the underline pixel row off when this toggle is in upper case state.

Examples:

S *Q* gives a question mark. ANS *Q* gives an upside-down question mark (as can be seen in appendix C). *Q* CAPS *Q* CAPS gives Qq (or qQ, depending on the initial state of the case shift toggle), and then restores the case shift.

Cursor keys



[INS] *Insert toggle*. The insert key toggles between the insert and overwrite state. In insert mode, the flashing cursor is a filled box, and in overwrite mode only a thin line. A characters entered in insert mode is inserted at the cursor position. In overwrite mode, the entered character replaces the character at the cursor position.

[DEL] *Delete*. This key is used to delete the character at the cursor position. When [S] [DEL] is pressed, the complete line at the cursor is deleted.

[⇧] *Cursor up*. The cursor is moved up one line with this key. [S] [⇧] is used to move the cursor to the first line of the text.

[⇩] *Cursor down*. The cursor moves down one line with this key. When [S] [⇩] is pressed, the cursor moves to the last line of the text.

[⇦] *Cursor left*. This key is used to move the cursor one character to the left. With [S] LTOP [⇦] the cursor moves to the first character in the line.

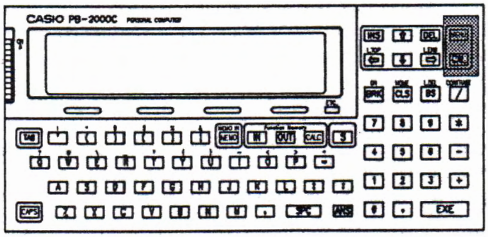
[⇨] *Cursor right*. To move the cursor one character to the right, this key is used. The cursor can be moved to the last character in the line by [S] LEND [⇨].

[CLS] *Clear screen*. In CAL mode, this key moves the cursor to the first character of the CAL area. To clear the CAL area, [S] HOME [CLS] has to be pressed.

[BS] *Backspace*. The character directly to the left of the cursor is deleted by this key. By pressing [S] LDEL [BS] the remainder of the line from (including) the cursor position is deleted.

The cursor keys are described in detail when the operating system is described, and where the cursor keys has a function.

Mode keys

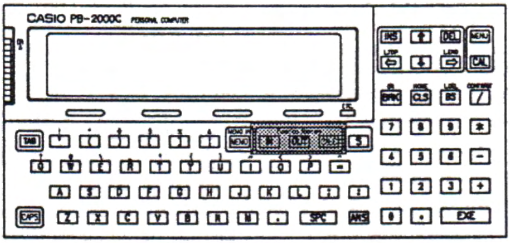


[MENU] *MENU mode*. This key selects the file MENU mode.

[CAL] *CAL mode*. The CAL mode is selected by this key.

The MENU and CAL modes are further described in chapters 6 and 7.

Function memory keys



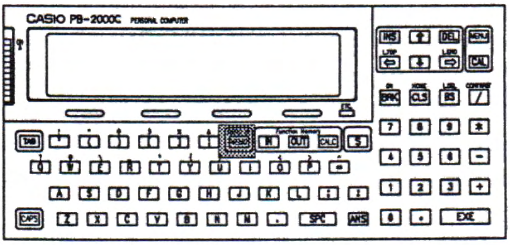
IN *Function memory input.* The contents of the line at the cursor is entered into the function memory when this key is pressed.

OUT *Function memory output.* When pressing this key, the function memory contents are inserted at the cursor position, according to the current insert state.

CALC *Function memory calculation.* This key is pressed to start calculation of the expression in the function memory.

The usage of the function memory is described in chapter 11.

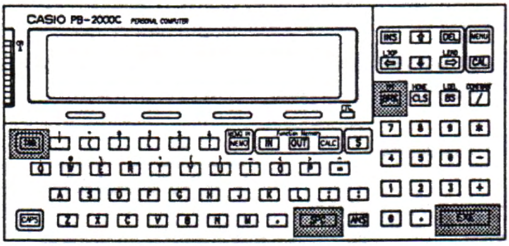
Memo note key



MEMO *Memo note key.* This key is used to search for data in the **MEMO** note pad. New data are added to the note pad by pressing keys S MEMO IN.

The **MEMO** note pad is described in chapter 10.

Special keys



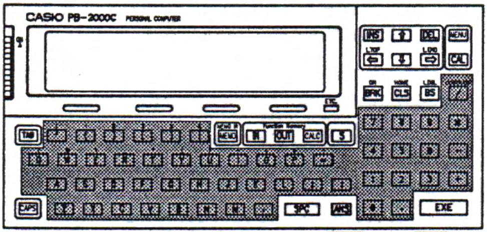
ON BRK *Break key.* By pressing this key, input wait from the cassette recorder or RS-232C is cleared. The key is also used to turn power on after an automatic power off. In the debugger, the program being run can be stopped by pressing this key.

EXE *Execute key.* This key is used to calculate an expression in **CAL** mode, and to run a program in **MENU** mode. In the editor, this key is used to begin a new line (like the carriage return on a typewriter).

SPC *Space bar.* A space is inserted when this key is pressed.

TAB *Tabulator*. With this key, a tabulation is inserted into a text. The default tabulator width setting is eight characters, but can be changed in the MENU mode.

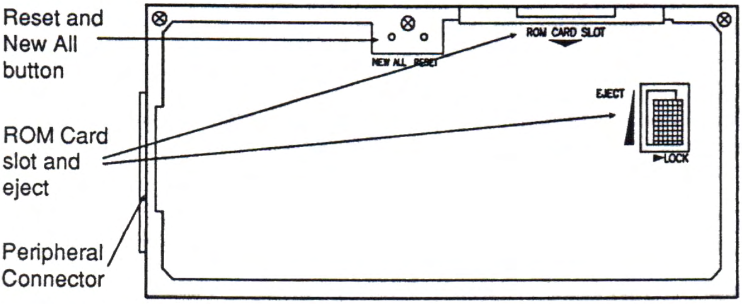
Other keys



The remainder of the keys are the alphabetic, numeric, and arithmetic operator keys. They are used to enter the symbol written on them.

The key CONTRAST / (called by pressing S CONTRAST /) is used to set the *contrast* of the display. After pressing this key, the keys ⇧ and ⇩ are used to increase or decrease the contrast, respectively. To end the contrast setting, press any other key (the sequence S S is sufficient and will not enter anything to the keyboard buffer).

PB-2000 Bottom view



○ RESET *Reset button*. This button is used to reset the computer. All programs and data are retained when this button is pressed, but the programmable keys and character generator are initialized to their default contents. It should only be used in emergency cases (e.g. the computer stops functioning due to an endless program loop), as the usage of this button has some draw-backs in some modes. These draw-backs are described where appropriate.

○ NEW ALL *New All button*. When this button is pressed in CAL mode, the memory of the PB-2000 is cleared. All programs and data are destroyed, and therefore the button should be used with care.

The connector is used to connect the FA-7 and MD-100 option units. These units are described in chapter 14.

External ROM cards are installed in the ROM card slot, and removed with help of the eject mechanism. Appendix B describes how to insert and remove ROM cards.

Precautions

There are a number of precautions that should be noted, in order to keep your PB-2000 and ROM cards in top condition:

- Do not try to disassemble the interior of the computer. If service is needed, contact the nearest CASIO dealer.
- Do not drop or use force on the computer.
- Do not expose the computer to high temperature, humidity, and/or dust.
- Do not connect foreign devices to the computer's connectors. Keep the connector covered when no device is connected, to avoid dirt and dust to enter the connector.
- Switch power off before connecting the computer to another device.
- Replace old batteries at least every second year, as old batteries may leak.
- Do not use thinner, benzine, or other volatile solvents to clean the computer. Use a cloth, either dry or dampened with water.

If the computer is exposed to low temperature, the response of the display will slow down or even halt. Normal operation of the display will resume when the temperature rises.

If the computer is exposed to strong static charge, the computer may stop operating. In such case, press **O** RESET to resume operation. If the memory contents have been destroyed, it might also be needed to press **O** NEW ALL.

Starting DL-Pascal

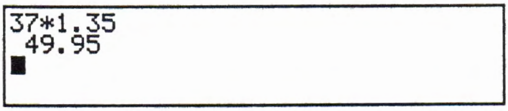
If you have DL-Pascal on a separate ROM card, this ROM card has to be installed in the PB-2000 before you can begin using DL-Pascal. The ROM Card slot is used for this purpose, and installation of a ROM card is explained in appendix B. This appendix also describes some error messages and considerations when changing ROM cards.

With the DL-Pascal ROM card installed (or built in), turn the power of the PB-2000 on. If everything is correct, the cursor will be flashing in the upper left corner of the display. If you have any problem getting power to the PB-2000, look in appendix A. This appendix explains problems regarding the power supply of the PB-2000.

The mode you are now seeing is called the **CAL** mode. It is usually used for calculations, but can also be used for other purposes. The **CAL** mode is normally called by pressing the CAL key.

To calculate the cost of 37 parts, each costing 1.35, you enter:

*37 * 1.35* EXE

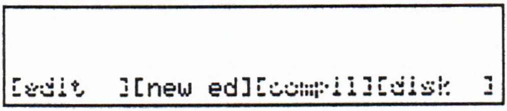


```
37*1.35
 49.95
■
```

The answer is displayed immediately: 49.95

The other main mode is the file **MENU** mode. Press the MENU key to get into this mode. You will see the following display (some text files may also be shown if this is not the first time you are using the computer):

MENU



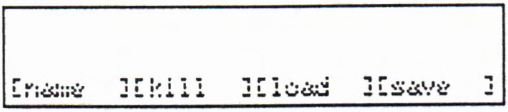
```
[edit   ][new ed][compil][disk   ]
```

The **MENU** mode is normally used when navigating through the files in DL-Pascal. From the **MENU**, files can be edited, compiled, loaded, saved, printed, removed, and much more.

The bottom line displays the current *function keys*. Function keys that cannot be used are dimmed. As the other function keys are dimmed, only `[new ed]` can be used at this moment.

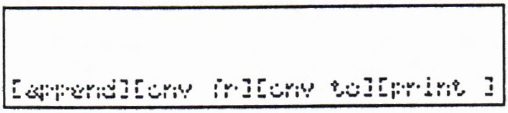
Often more than four menu choices exist. The other function key menus are retrieved by pressing the ETC. key (located to the right of the function keys). The **MENU** mode has a total of 16 function keys; four function menus, each with four choices.

ETC.



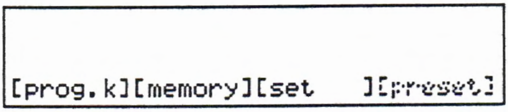
```
[name   ][kill   ][load   ][save   ]
```

ETC.



```
[append][cpy fr][cpy to][print ]
```

ETC.



```
[prog.k][memory][set    ][preset]
```

One more press on ETC. returns to the first function key menu.

A small note regarding the EXE key. It is used in the computer to signal that all characters have been entered for one line, and that execution should continue. Examples to this are: to calculate an expression (such as in the **CAL** area above), and when entering a file name on the function line. It is also used as a carriage return key in the editor. At single-key input, however, like answering *Yes* or *No* at file kill and pressing function keys in **MENU** mode, the EXE should not be pressed.

Now switch the power off. After a short while, switch the power on again. The display still shows the **MENU** mode, as it did when the power was switched off. This is the *power-on resume* function of the DL-Pascal ROM card. The program being run when the power was turned off is automatically resumed when the power is set on again, without losing anything!

After this introduction you are ready to write your first program. You will perhaps need to look back to this chapter to find some keys and the function of them.

Good luck with your program writing!

Chapter 3

Programming in DL-Pascal

The file menu system in the PB-2000 makes it easy to enter, compile and run programs. In this chapter we will write two small programs to show this. We will also demonstrate how to check the execution of a program using the debugger. This chapter is not intended to teach Pascal programming. If you need a book for learning Pascal, there are a number of books available to teach you this.

The Pascal text file entered is called the *source file*. In Pascal, the text can be entered in both upper and lower case; the compiler makes no difference between the case of the letters in the program. It is easier to enter programs in one case only, but using different casing helps when reading the program. All examples in this manual will use both lower and upper casing.

The source file is read by the *compiler*, which translates the program into an executable file, called the *code file*. The code file contains the executable code, debug information, and relocation information needed by the operating system when running the program.

A file is referenced by a *file specifier*. In the PB-2000, the file specifier contains a *name* and an *extension*, separated by a period. In addition, the system uses a *file type*. The name can be a sequence of up to eight characters, and the extension up to three characters. For a Pascal source file, the extension is normally **.PAS** (stands for **PAS**cal file). The extension is changed to **.EXE** for the code file (stands for **EXE**cutable file). The extension is actually not needed, but by using it in a consistent way, it helps to extract the files by function (e.g. all Pascal files). The file type is always **S** for text files (including source files), and is **E** for the code files. The file type cannot be manually changed, although it can be entered in the file specifier and used when creating a file.

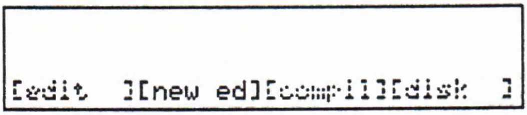
The *debugger* is used to verify that the program is working correctly (called debugging the program). In the debugger the program and it's data can be checked. Execution of the program can be stopped at specified points in the program. Program lines can also be skipped or re-executed. The program's data can be read and changed while being in the debugger.

During compilation, the compiler inserts *debug information* into the code file. This information is needed for the debugger to be able to debug the program. When the program is fully debugged and ready, there is no need to keep the debug information in the code file. At the end of the first program example we will show how to remove the debug information from the code file.

The First Program

Now let's start writing the first program. This is a small program that will write a text on the PB-2000's display.

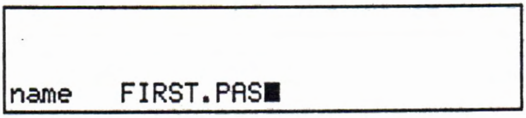
First, switch on the PB-2000 and set the **MENU** mode (press MENU if the **MENU** mode is currently not selected). Also make sure the first function menu is displayed at the bottom of the screen (if not, press the ETC. key to get to the right menu).



```
[edit  ][new ed][compil][disk  ]
```

To enter a new program file, press the `[new ed]` function key. This will first ask for the name of the new file. Let's call it **FIRST.PAS** (the extension **.PAS** for the Pascal source file).

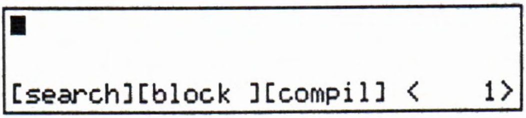
`[new ed]`
FIRST.PAS



```
name    FIRST.PAS■
```

After the file name has been given, the EXE key has to be pressed (as described on page 13). The editor is then automatically started.

EXE

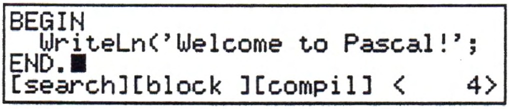


```
■

[search][block ][compil] <     1>
```

Enter the following program into the computer, and remember to press the EXE key to separate each line. If you make a mistake, press the BS key (this will delete the last character entered). Remember that you can enter the program in all lower or upper case if you want to.

```
PROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!';
END.
```

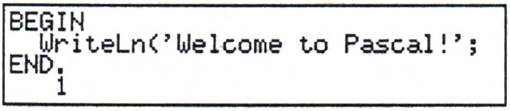
In the lower right corner of the display the current line number is displayed. It increments for each line terminated by the EXE key. Within a text line, the line number is the same, even if the text line occupies more than one line on the display (i.e. if there are 31 or more characters in the line).



```
BEGIN
  WriteLn(’Welcome to Pascal!’;
END.■
[search][block ][compil] <    4>
```

The program has to be compiled before it can be run. The compilation process converts the source file to a code file. The compiler can be called directly from the editor by pressing the `[compil]` function key. During compilation, a line number is displayed on the function menu line. This is the current line in the source file where the compiler is reading the program, and it is updated for each 16th line.

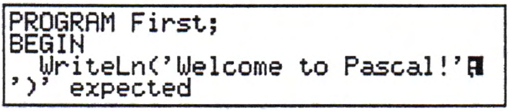
`[compil]`



```
BEGIN
  WriteLn(’Welcome to Pascal!’;
END.
    1
```

If any errors are found during compilation, the editor is reselected, and the cursor is positioned at the error, or at the identifier following the error. On the function line, an error message is also written to help you locate the error.

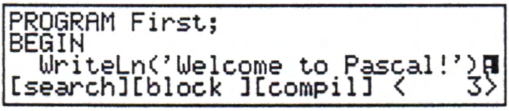
You should get a compilation error now, if you entered the program exactly as written, since there is a small error in it.



```
PROGRAM First;
BEGIN
  WriteLn(’Welcome to Pascal!’■
’)’ expected
```

As the error message says, a right parenthesis is missing. The Write statement must have the text to write enclosed in parentheses, and the right one is missing. Add it to the program (the cursor is placed exactly where the parenthesis is supposed to be).

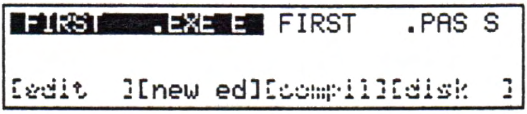
)



```
PROGRAM First;
BEGIN
  WriteLn(’Welcome to Pascal!’)■
[search][block ][compil] <    3>
```

Now, restart compilation by pressing the `[compil]` function key again. This time the program should be correct, and compilation continue to the end of the program. When compilation is successful, the computer returns to the **MENU** mode, with the cursor placed on the generated code file (**FIRST.EXE**).

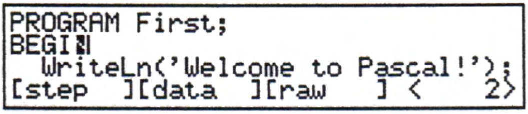
`[compil]`



```
FIRST    .EXE E FIRST    .PAS S

[edit   ][new ed][compil][disk   ]
```

To run the code file, just press the EXE key in **MENU** mode. This will start the debugger.

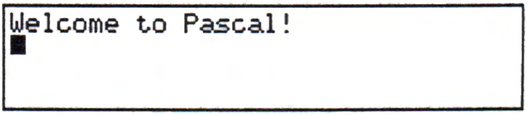
EXE



```
PROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
[step   ][data   ][raw    ] <    2>
```

If you just want to run the program (without using any functions of the debugger), press `[step]` and then `[go end]` function keys. The program will start executing directly and write the message on the display. Then program execution ends.

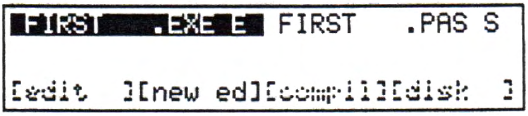
`[step] [go end]`



```
Welcome to Pascal!
█
```

When execution of a program ends, the **CAL** mode is automatically selected. In this way, the text contents of the display will remain on the screen. To return to the **MENU** mode, press the MENU key.

MENU

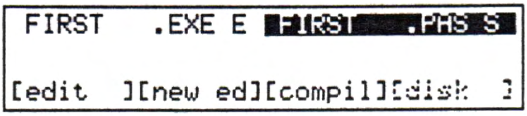


```
FIRST    .EXE E FIRST    .PAS S

[edit   ][new ed][compil][disk   ]
```

The program is now working, and there is no need to use the debugger any more. To make the program compile without putting any debugger information in the code file, the compiler option {$D–} is used. This compiler option controls the generation of debug information in the code file. When turned off (by the minus sign), no information about the source program or variables is stored in the code file.

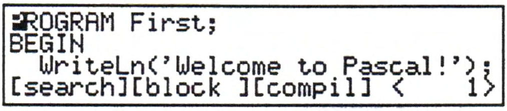
To add this compiler option to the program, move the cursor bar (in **MENU** mode) to the source file (**FIRST.PAS**), and press the `[edit]` key. This key is used to edit an existing program.

⇨



```
FIRST    .EXE E FIRST    .PAS S

[edit   ][new ed][compil][disk   ]
```

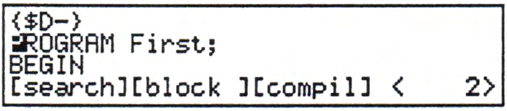
`[edit]`



```
▮ROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
[search][block ][compil] <    1>
```

Add the compiler option in the beginning of the program. A compiler option is always entered in a comment (surrounded by curly braces), with a dollar sign as the very first character in the comment and the compiler option directly following the dollar sign.

{$D-} EXE



```
{$D-}
▮ROGRAM First;
BEGIN
[search][block ][compil] <    2>
```

Compile the program again (by pressing the `[compil]` function key), and run the resulting code file by pressing the EXE key. This time the program runs directly, without calling the debugger.

Actually, it is possible to call a program containing debug information from the **MENU** mode, without going into the debugger. This is done by pressing S EXE when running the program, instead of just EXE.

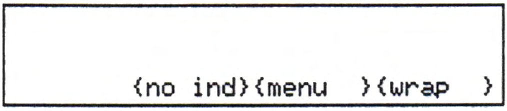
The Second Program

This program reads a name and writes it in reverse order. However, there are a few bugs in the program that must be located and corrected for the program to work. The debugger will be used for this purpose.

A separate procedure in the program is used to write the text in reverse order. This makes the main part of the program much easier to read.

Enter the following program in the same way as the first one. Call this program **SECOND.PAS**. If you want, the editor can automatically indent the lines according to the previous line. To do this, use the `[setup]` function key in the editor, as described on page 86.

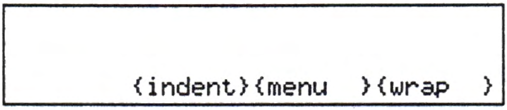
`[new ed]`
SECOND.PAS EXE
ETC. `[setup]`



```



        {no ind}{menu   }{wrap   }
```

`{edit}`

EXE ETC.



```



        {indent}{menu   }{wrap   }
```

At a new indentation level, you will still need to insert the wanted number of spaces, or delete some spaces by the BS key.

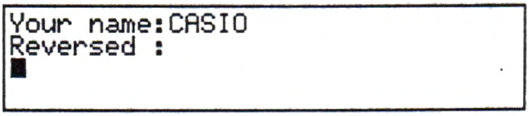
Now, enter the program:

```
PROGRAM Second;
PROCEDURE WriteRev(str:STRING);
  VAR len,i:Integer;
  BEGIN
    len:=Length(str);
    FOR i:=len TO 0 DO
     Write(str[i]);
  END;
VAR s:STRING;
BEGIN
  Write('Your name:');
  ReadLn(s);
  Write('Reversed :');
  WriteRev(s);
  WriteLn;
END.
```

Now compile the program (use the `[compil]` function key). No compilation error should occur. (If it does, correct the error against the program listing, and recompile the program.)

Let's run the program. Start the code file from the **MENU** mode by pressing the EXE key, and run the program directly by pressing `[step]` `[go end]` function keys. Enter *CASIO* as the name and press EXE. The name should now be written in reverse order.

EXE
`[step]` `[go end]`
CASIO EXE

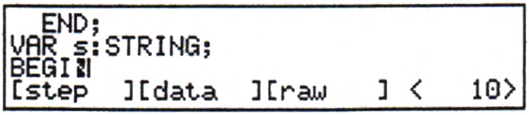


```
Your name:CASIO
Reversed :
█
```

Obviously there is an error in the program. We will now try to find this error.

As the program execution has ended, we are now in the **CAL** mode. To start the program again, go back to **MENU** mode (the MENU key), move the cursor to the right code file (**SECOND.EXE**), and press EXE to go into the debugger

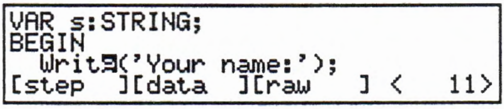
MENU ⇨ EXE



```
  END;
VAR s:STRING;
BEGIN
[step  ][data  ][raw   ] <   10>
```

The debugger screen displaying the code is called the *code window*. We will now step through the first statements of the program to see that the name has been read correctly. To step to the next statement in the code, you press the function key `[step]` followed by function key `[go nxt]`. This will place the cursor at the first **Write** statement.

`[step] [go nxt]`

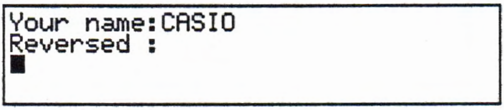


```
VAR s:STRING;
BEGIN
  Writ█('Your name:');
[step  ][data  ][raw   ] <   11>
```

Press `[step] [go nxt]` once more. You will see the display flash the written text briefly, and then the cursor will be placed on the **ReadLn** statement in the code window. If you want to see your program's display (the *output window*), you can press the function key under the line number. The output window is then shown. A press on any key will return to the code window.

`[step] [go nxt]`
`< 12>`

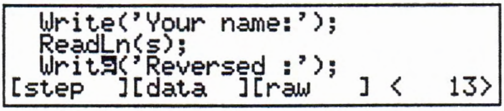
EXE



```
Your name:CASIO
Reversed :
█
```

Step to the next statement by pressing `[step] [go nxt]`. This time the program will wait for you to input the text. Enter *CASIO* again and press EXE. The code window will once again be displayed.

`[step] [go nxt]`
CASIO EXE

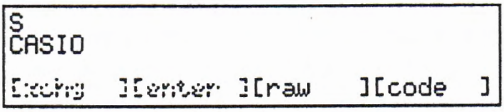


```
  Write('Your name:');
  ReadLn(s);
  Writ█('Reversed :');
[step  ][data  ][raw   ] <   13>
```

Has the text been read correctly? You can look at the program's variables by pressing the `[data]` function key. The *data window* is then displayed. As variable **s** is the first global variable, it will be the first one to be displayed.

`[data]`

`[code]`



```
S
CASIO

[chng  ][enter ][raw   ][code  ]
```

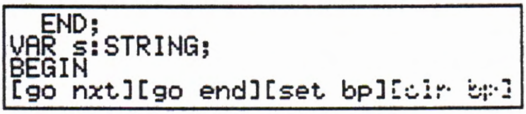
That looks all right. Go back to the code window by pressing `[code]` function key.

Obviously, the error is in the **WriteRev** procedure. Stepping to the next statement will call any procedures without stepping through them. As we want to look at variables inside of the procedure, statement stepping cannot be used in this case. We will therefore run the program until the procedure has been entered. In addition, we will set a break point at the end of the procedure, so we can look at the variables before execution returns from the procedure. As long as we are in the debugger, execution will always stop at a break point.

Move the cursor to the END statement in the procedure (on line 8). The cursor has to be placed somewhere in the first word of the statement (i.e. on the E, the N, or the D character) to be able to set the break point. The break point is set by pressing function keys `[step] [set bp]`.

⇧ ⇧ ⇧ ⇧ ⇧
⇦ ⇦ `[step]`

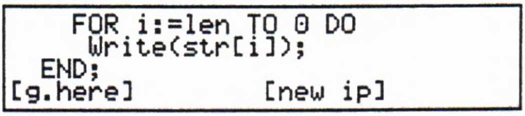
`[set bp]`



```
  END;
VAR s:STRING;
BEGIN
[go nxt][go end][set bp][clr bp]
```

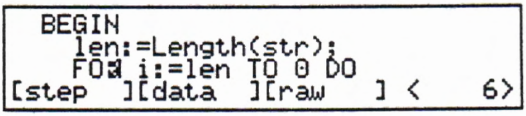
The break point has now been set. Next job is to execute to a statement in this procedure. A good place is the second statement, where we can see that the string length has been read correctly. Move the cursor to this statement (the statement beginning with FOR on line 6). Place the cursor anywhere on the word FOR. To execute until this statement is reached, the `[step]` function key is first pressed. There is a second menu at this key, so press the ETC. key to get to it. Last, press `[g.here]` to execute up to this statement.

⇧ ⇧
`[step]` ETC.



```
    FOR i:=len TO 0 DO
     Write(str[i]);
  END;
[g.here]          [new ip]
```

`[g.here]`

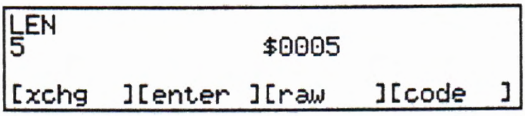


```
  BEGIN
    len:=Length(str);
    FOR i:=len TO 0 DO
[step  ][data  ][raw   ] <    6>
```

You can now look at the variables in the data window (press the `[data]` function key). The first variable is the string parameter, and that looks all right. The next variable is selected by pressing the ⇩ key. This is variable i, and it's contents may look a bit funny. This is because the variable has not yet been assigned any value. Press ⇩ again to get to variable len. This variable has the length of the string, and a value of 5 is surely correct. Press `[code]` to get back to the code window.

`[data]` ⇩ ⇩

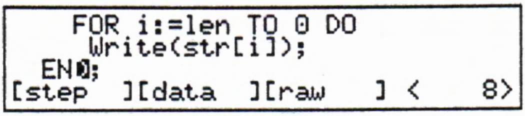
`[code]`



```
LEN
5                $0005

[xchg  ][enter ][raw   ][code  ]
```

Step to the next statement by pressing `[step] [go nxt]`. The statement is the Write statement in the FOR loop.

`[step] [go nxt]`

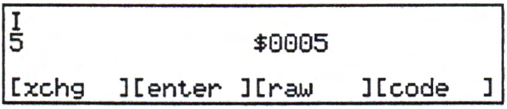


```
    FOR i:=len TO 0 DO
     Write(str[i]);
  END;
[step  ][data  ][raw   ] <    8>
```

Hmm, that is not the right statement. The Write statement is skipped completely. Let's look at variable i in the data window, by pressing `[data]` ⇩. The value is 5, which is the initial value of the loop. That should be zero if the loop had executed correctly. Press `[code]` to go back to the code window.

`[data]` ⇩

`[code]`



```
I
5                    $0005

[xchg  ][enter ][raw   ][code  ]
```

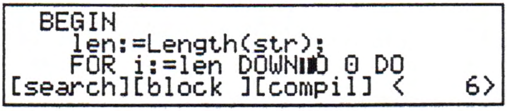
The error is obviously in the loop. It should count backwards from the length of the string down to zero. Can you find the error?

By mistake, the direction of the loop was set forward by the keyword TO, instead of DOWNTO for backward direction. That is the error, and we will correct this error now.

To correct any errors, the editor must be used. The easiest way to do this is to first go to the **MENU** mode by pressing MENU key. This will end the debugger and also place the cursor on the source file last displayed in the debugger. Then just press `[edit]` to get to the editor.

Move the cursor to the T in TO on line 6, and add *DOWN* to it.

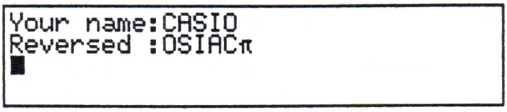
MENU `[edit]`
⇩ ⇩ ⇩ ⇩ ⇩
⇨....
DOWN



```
 BEGIN
   len:=Length(str);
   FOR i:=len DOWN█O 0 DO
[search][block ][compil] <    6>
```

Press `[compil]` to compile the program, and then run the program by pressing the EXE key. Run the program to the end by pressing function keys `[step]` `[go end]`. Re-enter *CASIO* as the name, and watch OSIAC being written.

`[compil]`
EXE
`[step]` `[go end]`
CASIO EXE



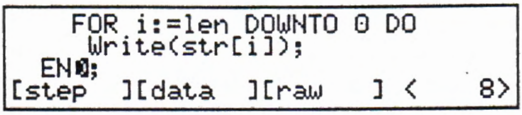
```
Your name:CASIO
Reversed :OSIACπ
█
```

Except for the additional pi, the output is correct. Where did this pi come from?

Start the debugger again, using the **MENU** mode. Place the cursor at the END statement of the procedure (line 8), and execute the program up to this statement. This is done in the same manner as the last time you used the debugger. If you have any problems, look on page 22.

Enter *CASIO* as the name at the input message. When the code window is displayed, look at the output window (by pressing the line number function key). The pi is display this time also. Press a key to return to the code window.

MENU ⇨ EXE
⇧ ⇧ [step]
ETC. [go end]
CASIO EXE

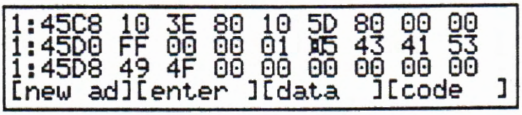


```
   FOR i:=len DOWNTO 0 DO
     Write(str[i]);
   END;
[step  ][data  ][raw    ] <    8>
```

In the FOR loop, the last character to be written is the zeroth element of the string. Look at the string in the data window (press `[data]`). The string looks correct, with the C as the first character. But in Pascal, the zeroth element of a string is the length byte! This is invisible in the data window (it is only used to get the number of characters to write). In order to look at this length byte, you have to look at the variable at a lower level. This is one use of the *raw window*.

Press the `[raw]` function key to get to the raw window. When the raw window is called from the data window, the cursor will be placed on the first byte of the variable that was displayed in the data window, and the address of the variable can be seen by the position of the cursor (the actual address of this variable, and the value of surrounding bytes might be different from what you will get).

[data]
[raw]

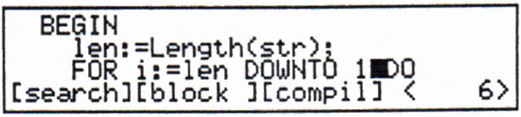


```
1:45C8 10 3E 80 10 5D 80 00 00
1:45D0 FF 00 00 01 05 43 41 53
1:45D8 49 4F 00 00 00 00 00 00
[new ad][enter ][data   ][code   ]
```

This is the length byte of the string. The length of the string is five characters, and if you look in appendix D you will see that the code for the pi character is also 5. The loop writes the length byte in addition to the actual string!

Go back to the code window (press `[code]`). The last element to write should be element 1 of the string (not zero as that is the length byte). Adjust the FOR loop so the last value is 1 instead of 0. Remember, you must be in the editor to be able to make any changes to the program. Place the cursor on the zero, press DEL to remove it, and press the digit 1.

[code] MENU
[edit]
⇩ .... ⇨ ....
DEL *1*



```
 BEGIN
   len:=Length(str);
   FOR i:=len DOWNTO 1 DO
[search][block ][compil] <    6>
```

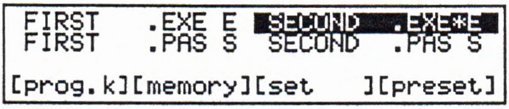
Now compile and execute the program. This time the result is correct.

If you want to call the program a number of times, it can be a bit tedious to constantly go back to the **MENU** mode and locate the file to run the program again. To avoid this, it is possible to specify up to four programs that can be run directly from the **CAL** mode, just by pressing one of the function keys. This is called the *preset function*.

To set the preset mode on this code file, go back to the **MENU** mode, and move the cursor to the code file (**SECOND.EXE**). Press the key ETC. three times, until the fourth function key choice reads `[preset]`. Press that function key, and an asterisk will be shown after the file extension. This signals that the preset function has been set on this file.

MENU ⇒ ETC. ETC. ETC.

`[preset]`



```
 FIRST    .EXE E  SECOND    .EXE*E
 FIRST    .PAS S  SECOND    .PAS S

[prog.k][memory][set    ][preset]
```

Select **CAL** mode by pressing CAL. At the first function key, the choice `[SECOND]` will be visible. Press this function key, and the program will be run directly, without calling the debugger. When execution ends, the function key will be shown again.

The preset function will remain connected to the file until it is removed by another press on `[preset]`, or when the file is removed or recompiled.

Chapter 4

Separate Compilation

One of the most useful features of DL-Pascal is separate compilation. It allows the creation of library units that can be directly used by different programs. The units are compiled in advance, so there is no need to reveal the source code of the units. In addition, one and the same unit can be used by different program, but only one copy of that unit is needed. This will keep memory consumption low.

Thanks to separate compilation, it is possible to make and sell library units that can be used by other persons in their programs, but without being able to look at the source code of the unit.

Two source files are needed for a separate compiled unit. The first file is the *interface* unit, where types, variables, procedures, and functions are declared, that can be used by other programs. Procedures and functions are only declared by their heading (i.e. their name and parameters) in the interface unit.

The second file is the *implementation* unit, which is similar to a normal program. Here, the actual implementation of the procedures and functions is entered. The implementation unit can be changed at any time, without any need to recompile programs using that unit. Recompilation of a program is only needed if an interface unit used by the program is changed, e.g. by introducing new parameters in a function or procedure. There is a *checksum* stored in the interface and implementation units, and a corresponding checksum in all programs using that unit. This checksum is used to detect a changed interface unit, before the program is run. If this could not be detected, program execution would not be made correctly, and the memory contents presumably be destroyed.

The name of the unit's source files is the first eight characters of the unit's name, in upper case. The upper case is needed as the Pascal compiler internally converts all identifiers to upper case, and that converted name is then used to locate the unit files. The extension of the interface source file should be **.INT**, and the implementation source file can be **.PAS**. Both files need to be compiled. After compilation, the interface code file will have extension **.UNI** (for **UNI**t) and file type **U**, and the implementation code file will have extension **.DLE** (for **D**ynamic **L**ink **E**xecutable) and file type **E**.

In this chapter, we will write a small benchmark program. This program uses another unit to write the opening message, and the start and stop timing messages. We will also show how easy it is to modify the implementation unit without needing to recompile the programs using that unit.

Finally, we will write one additional program using the same unit for another benchmark. By thinking about this before writing the unit, we will be able to write this second program without having to change the unit (more than adding a opening text message).

Creating the Unit

Before writing the unit, we must think about which procedures are needed in the unit. It is fully possible to add or change the declarations of the interface unit after it has been written (in a normal program project this has to be made a number of times), but it helps to be as thorough as possible the first time.

We need a unit to write the texts of some benchmark programs. There is a need to write a title, possibly with a version number and a text describing what the program will test. The title and version number can be the same for all programs, but the text will differ. In this case, we will use a number for that text. In this way, we can later change the actual text in the unit.

We also need one procedure for the start message of the benchmark, and one for the stop message. None of these procedures need any parameters.

As a result of these thoughts, the interface unit can have the following definition:

```
INTERFACE BenchText;
PROCEDURE Init(textNr:Word);
PROCEDURE Start;
PROCEDURE Stop;
END.
```

Enter this file in the editor, by using the name **BENCHTEX.INT** (the first eight characters of the unit name in upper case). The interface file can also be compiled at this time; the implementation unit is not needed to compile the interface unit.

Next, either the implementation of the unit, or the actual program can be entered and compiled. When compiling a program, only the interface code files are needed. At run time, however, the implementation code file must be available, but the interface file doesn't. By saving the interface code files before running a program, more memory can be made available to the program.

We will first write the source file of the implementation unit. This file consists of the procedure bodies. In both the Start and Stop procedures, a message is written. A sound is then generated to make an acoustic signal of the starting and stopping of the benchmark. Last, a key press is awaited. The Stop procedure also clears the screen.

The implementation file will have the following look:

```
UNIT BenchText;
USES Crt;
PROCEDURE Init(textNr:Word);
  BEGIN
    ClrScr;
    WriteLn('    Benchmark program ver 1.0');
    CASE textNr OF
    1: WriteLn('    FOR-loop: from 1 to 10000');
    END;
  END;
PROCEDURE Start;
  VAR ch:Char;
  BEGIN
    GotoXY(1,4);
    Write('Start at key press');
    ch:=ReadKey;
    GotoXY(6,4); ClrEol;
    Sound(500,100);
  END;
PROCEDURE Stop;
  VAR ch:Char;
  BEGIN
    Sound(1000,200);
    GotoXY(10,4);
    Write('Stop');
    ch:=ReadKey;
    ClrScr;
  END;
BEGIN
  {init}
END.
```

The implementation unit file is now entered, using the name **BENCHTEX.PAS.** After entering the file, compile it. The implementation unit will need the interface unit code file. If that code file is not available, you will notice that a compilation error is generated at the start of the implementation unit compilation.

The main part of the implementation unit (with the {init} comment above) is used for initialization. This part of the unit is always called before the main part of the program using this unit is called.

The BenchText unit also uses another unit (the predefined unit Crt). A unit can use other units without problems. In one special case, however, special precautions must be taken. This is explained on page 187.

Writing the Program

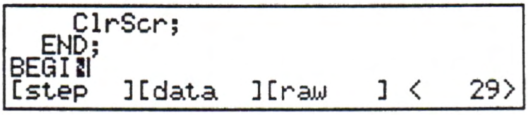
As you could see by the texts in the implementation unit, the first program using this unit will be a benchmark test of an empty loop, running from 1 up to 10000. The actual program is quite neat, as most of the operations regarding input and output of the benchmark program were made in the unit:

```
PROGRAM BenchFor;
USES BenchText;
VAR i:Word;
BEGIN
   Init(1);
   Start;
   FOR i:=1 TO 10000 DO;
   Stop;
END.
```

That program is easy to follow. Enter and compile the program (let the name be **BENCHFOR.PAS**).

This program can now be run directly. Press EXE from the MENU mode to start the debugger.

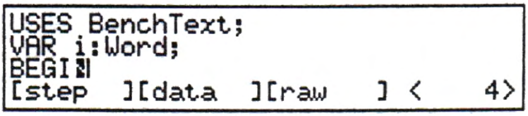
EXE



```
      ClrScr;
   END;
BEGIN
[step   ][data   ][raw    ] <   29>
```

The main program uses unit BenchText. Since the initialization code of a unit is called before the program's main part, that unit is the first one to be shown in the debugger. If you press key ↓, you will see the {init} comment. We will not debug this unit, so press `[step]` `[go end]` to continue to the actual program. When the debugger is in the main part of a unit, that key combination will just execute up to the main part of the next unit, or to the main part of the actual program.

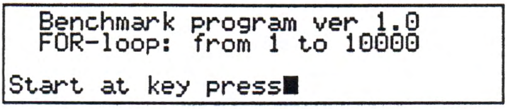
`[step] [go end]`



```
USES BenchText;
VAR i:Word;
BEGIN
[step   ][data   ][raw    ] <    4>
```

Now the actual program is shown in the debugger. Let's run the program (we will not use the debugger) by pressing `[step] [go end]`.

`[step] [go end]`



```
  Benchmark program ver 1.0
  FOR-loop: from 1 to 10000

Start at key press■
```

To start the benchmark, be prepared with a stop watch and press [EXE]. As you surely noticed, the program execution is *fast*. Try the same type of benchmark using interpreting BASIC or C, and you will see the difference.

When we executed the program, the BenchText unit was first displayed in the debugger. Since that unit (supposedly) works, we do not want to have the debugger stop at this unit. We will therefore change the implementation of that unit, so no debug information is entered in the code file. As in the previous chapter, the compiler directive {$D–} is used.

Edit the source file **BENCHTEX.PAS**, so the program will begin with:

```
{$D-}
UNIT BenchText;
...
```

Compile the source file by pressing the `[compil]` function key, and a new implementation code file will be generated.

As only the implementation source file has been changed, the program can be run immediately. There is no need to change or compile any programs using the unit. Move the cursor bar to the program code file **(BENCHFOR.EXE)** and run it by pressing [EXE]. The debugger will be invoked, but this time the program source file will be displayed immediately. The unit file is skipped, thanks to the compiler directive.

Writing Another Program

We will reuse the unit with another benchmark program. This benchmark will call an empty procedure 10000 times. The benchmark program looks like this:

```
PROGRAM BenchCall;
USES BenchText;
VAR i:Word;
PROCEDURE CallIt;
  BEGIN
  END;
```

(continued)

(continued)

```
  BEGIN
    Init(2);
    Start;
    FOR i:=1 TO 10000 DO CallIt;
    Stop;
  END.
```

This program can be entered in the editor and also be compiled. Use the name **BENCHCAL.PAS**. As you might have noticed in the program, we are calling the Init procedure of the BenchText unit with a new text number. If you run this program now, no subtitle will be written. We have to add this text to the implementation unit.

This addition is easy to make. Move the cursor bar (in **MENU** mode) to the implementation file **BENCHTEX.PAS**, and press `[edit]` to edit the file. Move the cursor down to the CASE statement in line 8, and place the cursor at the end of line 9 (the first case selector). Press EXE to add a new line, and enter the new line so the program reads:

```
  ...
      1: WriteLn('   FOR-loop: from 1 to 10000');
      2: WriteLn('  FOR-loop calling a procedure');
      END;
  ...
```

Now compile this changed implementation unit. Still, the interface unit has not been changed, so the first program can still be run without any problems.

Run the newly created program's code file (**BENCHCAL.EXE**). You will see the new text being written in addition to the other, well-known, texts. Be prepared with a stop watch, and press EXE. This program is not as fast as the other one, but then this program has to do a lot of more work. You can also try this type of program in BASIC or C. It is still a lot of difference in execution time!

The implementation unit can be changed a lot without any need to recompile the other programs, as long as the declarations of the interface unit remain the same. For example, you can translate all texts in the unit to another language. Both programs using the unit will then be translated at the same time! By having all texts in a separate unit, it is easy to translate the program to another language. It can even be done without having the source code for any other unit or program (except the unit where the texts are entered, of course). When a program using the text unit is executed, the changed text unit is automatically used.

Part 2

Operating System

Chapter 5

Introduction

This part of the manual describes the operating system of the PB-2000. The various operation modes are described, and also the facilities possible with input line editing and programmable keys.

The two normal operation modes are the **MENU** and **CAL** modes. In *MENU mode*, the files in the PB-2000 can be controlled. This mode displays all files (up to six at a time), and the files can be edited, compiled, saved, loaded, etc., directly from this menu.

The *CAL mode* is mainly used for calculations, but can also be used in command-line mode. From the command line, functions can be executed by entering a command directly from the keyboard, in the same way as on a large PC.

There are also two special modes: the **MEMO** note pad and the **Function Memory**. In the *MEMO note pad*, text and data can be entered. Later, the entered data can be searched directly from the **CAL** mode. The *Function Memory* is used for repetitive calculations of one or more expressions.

The *Editor* is a full-screen text editor. It is used to enter text and program source files. Among the functions included in the editor are search-and-replace and block handling.

A Pascal program is compiled using the *Compiler*. To verify that the compiled program is correct, the *Debugger* is then used. These two functions are repetitively used until the program works as expected.

Programmable Keys

There are 20 *user programmable keys* in the Pascal card. Each key can be programmed with a sequence of up to 23 characters. All key codes can be entered to a programmable key.

The lower two alphabetic key rows, when used with the S, can be programmed (keys S *A* through S *:* and S *Z* through S SPC). Programming of these keys can be done in **CAL** and **MENU** modes (see chapter 6 and 7). A Pascal program can also program these keys (as described on page 294). When the **○ RESET** button is pressed, the default contents of the programmable keys are restored.

The programmable keys have the following default contents (the numbers within parentheses are used by the DEFKEY command in CAL mode):

A (0): LABEL EXE EXE
S (1): CONST EXE EXE
D (2): TYPE EXE EXE
F (3): VAR EXE EXE
G (4): PROCEDURE EXE
H (5): FUNCTION EXE
J (6): ARRAY
K (7): STRING
L (8): RECORD EXE END; EXE
; (9): INTERFACE
: (10): UNIT
Z (11): FOR TO DO EXE
X (12): IF THEN EXE
C (13): REPEAT EXE EXE UNTIL EXE
V (14): WHILE DO EXE
B (15): BEGIN EXE EXE END; EXE
N (16): RUN
M (17): DEBUG
, (18): DO .BAT
SPC (19): EDIT

Input Line Edit

When a line of text is entered on the function line as the response to a function key, the contents of that line can be edited. This is the *input line edit* function. This editing is also used for input to a Pascal program.

If the computer prompts with a default name, that name can be edited if a non-character key is the first key pressed (e.g. cursor keys). If a character key is pressed as the first key, the default name is removed and the input line is cleared. You can then enter a completely new name.

If insert mode is set (the cursor is a flashing box), new characters are inserted at the cursor position, and the remainder of the line pushed forward. If insert mode is off (the cursor is a flashing line), new characters will replace the character at the cursor position.

Within the line, the following keys have special functions:

⇦	Move cursor one character to the left.
⇨	Move cursor one character to the right.
S L.TOP ⇦	Move cursor to the first character in the input line.
S L.END ⇨	Move cursor to the last character in the input line.

DEL	Delete the character under the cursor.
S DEL	Delete the complete input line.
BS	Delete the character to the left of the cursor.
S L.DEL BS	Delete the remainder of the line (from the cursor position).
IN	Copy the input line to the function memory (described in chapter 11).
OUT	Insert the contents of the function memory at the cursor position.
INS	Toggle the insert flag. If the flag was set for insert mode, it will change to overwrite mode, and vice versa.
S INS	Insert the next character without decoding. This makes it possible to insert control characters on the line.
ON BRK	Ends line input and returns without executing the function asking for the input line.
EXE	Ends line input in a normal way, executing the specified function.
S EXE	Ends line input as with EXE . At programmable keys input, the previous key will be displayed after this key.
MENU	Ends line input as with ON BRK, but sets **MENU** mode.
CAL	Ends line input as with ON BRK, but sets **CAL** mode.
MEMO IN MEMO	Ends line input as with ON BRK, but sets **MEMO** input mode.

File Specifier Structure

The *file specifier* is used to name a file and/or device for usage. The specifier in the PB-2000 has four parts: the *device name*, the *file name*, the *extension*, and the *file type*. When entering a file specifier, some fields can be omitted.

```
dev:filename.ext;t
```

The file name and extension can use the characters * and ? as wild cards. The question mark will match any character in that position. The asterisk can only be used at the end of the file name and/or extension, and will match all files matching up to that character position.

Device Name

The device name is terminated by a colon. If the device is not entered, the internal RAM is normally selected (this is specially noted if it is not the case). Lower and upper case letters are treated as equal letters.

The following device names are allowed, with the corresponding device described:

0:	Floppy disk
A:	Floppy disk
AUX:	RS-232C
C:	Internal RAM memory
CAS:	Cassette
COM:	RS-232C
CON:	Console (display and keyboard)
LPT:	Parallel printer
PRN:	Parallel printer

For devices having more than one device name, either name can be used. They both denote the same device.

File Name

Files are named by the file name. A sequence of up to eight letters and symbols can be used. Lower and upper case letters are distinct letters in the file name. Spaces are used to fill the remainder of the name if less than eight characters are entered. All symbols except the following ones can be used:

. : ; < > ? *

When used in unit file names, all letters should be upper case. This is needed because the compiler internally converts all letters to upper case (including unit name letters), and therefore cannot find a file that has lower case letters in the name.

Extension

The extension part of a file specifier is normally used to mark the usage of the file. The file name is used to set the name of the contents of the file, and the extension is the group name of similar files.

An extension is started by using a period after the file name. If the extension is not entered, spaces will be automatically set as the extension.

Some extensions are used by the system and/or have a standard meaning:

BAT	Batch files
DLE	Compiled implementation unit file
EXE	Compiled program file
INT	Interface unit file
PAS	Program or implementation unit file
UNI	Compiled unit file

File Type

The file type is used by the system to control operations on various files. It is normally not used by the user, but can be entered by using a semicolon just ahead of the file type.

The following file types are possible with the DL-Pascal ROM card:

E	compiled program or implementation file
S	normal text of program (sequential) file
T	type-declared Pascal file
U	compiled interface file

Printer File Name

The file name used with the printer device names (PRN: and LPT:) is usually blank. One letter can be entered, however, to specify conversion of the codes sent to the printer. If omitted, no conversion will be made.

Conversion is specified when one of the following letters (in upper or lower case) are entered as the first letter:

C	Canadian-French IBM character set
D	Danish-Norwegian IBM character set
E	Epson 7-bit printer
I	Normal IBM character set
P	Portuguese IBM character set

Conversion is done to the nearest possible character in the destination character set.

RS-232C File Name

With the RS-232C device, the file name is used to specify the *communication protocol.* Seven letters in the file name are used, and the three letters in the extension. In this case, the extension is not used to specify the file group.

The structure of the RS-232C file name and extension is as follows:

```
rpbstdc.xov
```

The file name and extension can be entered in either upper or lower case. If a position is to be omitted, either a space or a non-allowed character must be entered. For such omitted positions, the previous specification is kept. If no file name or extension is entered at all, no change of the previous specification is made. The conversion protocol must, if it is wanted, always be specified.

The meaning of the positions of the file specifier is:

R (Bits per second rate)

The communication speed is given in bits per second. A number code is entered to set the speed. At 9600 bps, the computer might not be able to receive data without overflowing the internal buffer.

0	75 bps
1	150 bps
2	300 bps
3	600 bps
4	1200 bps
5	2400 bps
6	4800 bps
7	9600 bps

P (Parity)

This entry is used to specify the parity bit. The letters allowed are:

E	even parity
N	no parity
O	odd parity

B (Data bits)

The number of data bits can be specified as either 7 or 8 data bits. Only 8 data bits can be used when the communication speed is 9600 bps.

7	7 data bits
8	8 data bits

S (Stop bits)

The number of stop bits are specified here. Either 1 or 2 stop bits can be entered. When the communication speed is set to 9600 bps, 2 stop bits should be used.

1	1 stop bit
2	2 stop bits

T (CTS control)

CTS input signal is used for handshaking, and can be disabled by this entry. The following letters are allowed:

C	CTS checking on
N	no CTS checking

D (DSR control)

Handshaking through the DSR input signal can be disabled here. This is controlled through the letters:

D	DSR checking on
N	no DSR checking

C (CD control)

The input signal CD can be disabled by this entry. The allowed letters are:

C	CD checking on
N	no CD checking

X (XON/XOFF busy control)

Software handshaking normally uses the XON/XOFF protocol. When this is in use, characters $11 and $13 are used for handshaking and cannot be used for data. The letters to specify XON/XOFF are:

B	XON/XOFF used
N	no software handshaking

O (SI/SO protocol)

This protocol is used together with 7 data bits to specify the eighth data bit. The letters allowed for this field are:

S	SI/SO used
N	not used

V (Conversion protocol)

The conversion protocol is specified as in the printer file name. The conversion used is depending on whether the characters are sent or received. When characters are sent to the RS-232C device, conversion is done to the character set specified. When characters are received, conversion is done from the character set to the ISO-8 character set used by the PB-2000.

Examples:

```
7N82NNN.NN
4E71CDC.XOI
```

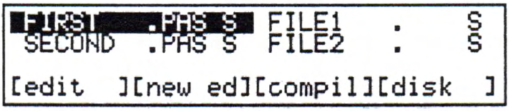
The first example will use 9600 bps with no parity, 8 data bits, and two stop bits. No handshaking will be used (neither hardware nor software) and no conversion will be made.

The second example uses 1200 bps with even parity, 7 data bits, and one stop bit. Both hardware handshaking (CTS, DSR and CD) and software handshaking (XON/XOFF) is selected. Conversion will be made to the IBM character set (this character set is used at the other end of the RS-232C cable).

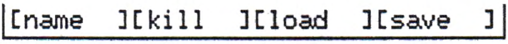
Chapter 6

MENU Mode

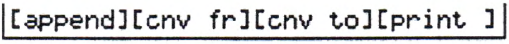
The **MENU** mode is used for easy operations on the files in the PB-2000. Up to six files are shown in the display at one time. From the **MENU**, 16 functions can be called by pressing a function key. The function keys allow the files to be handled in an easy way. The function line at the bottom line of the display shows the current contents of the function keys.



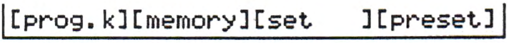
```
FIRST    .PHS S  FILE1    .    S
SECOND   .PHS S  FILE2    :    S

[edit   ][new ed][compil][disk   ]
```



```
[name   ][kill   ][load   ][save   ]
```



```
[append][cnv fr][cnv to][print ]
```



```
[prog.k][memory][set    ][preset]
```

The different menus are selected by pressing the ETC key. The first menu can always be selected by pressing S ETC.

A menu choice can be dimmed, in which case that function key cannot be used with the currently marked file.

If an error occurs while executing a function, the function is terminated and an error message is displayed on the function line. This message is removed by pressing a key.

Selecting a file

The current file is displayed by the cursor bar. This *selected* or *marked* file is the one operated on by the function keys.

The cursor bar can be moved around the files by the following keys:

⇨	Move the cursor bar to the next file.

⇦	Move the cursor bar to the previous file.
⇩	Move the cursor bar to the file below this file.
⇧	Move the cursor bar to the file above this file.
S ⇩	Move the cursor bar to the last file.
S ⇧	Move the cursor bar to the first file.

The **CAL** mode can be selected from the **MENU** mode by pressing the CAL key. Also, the **MEMO IN** mode can be selected by S MEMO IN/MEMO.

The letter keys can also be used in the **MENU** mode. Pressing letter keys select only the files beginning with the typed letters. This *file selection* feature is very useful when a large number of files exist.

A *wild card* character can be entered by pressing the question mark key. At such character positions, any character in the file name is matched.

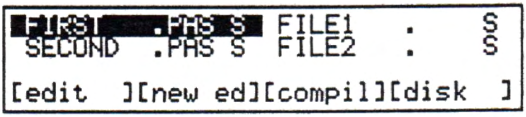
The key BS is used to delete the last character typed. The letters typed are not shown, but the files displayed are changed immediately. The **MENU** mode key is used to display all files again. In this case, the cursor bar will still be pointing to the file currently marked.

The period key is used to select the extension part of the file specifier. The BS key will then move from the extension to character number eight in the file name.

Please do remember that upper and lower case letters are treated as different letters in file names. If no files are displayed when a letter key is pressed, it may be needed to use the MENU and CAPS keys to remove previously typed letters and change the casing of the letters being typed.

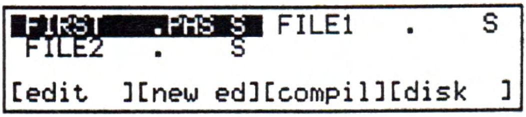
Example:

From the following list of files, all files beginning with **F** are selected:



```
FIRST   .PAS S  FILE1    .   S
SECOND  .PAS S  FILE2    .   S

[edit  ][new ed][compil][disk  ]
```

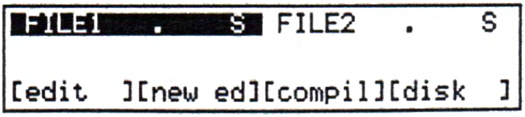
F



```
FIRST   .PAS S  FILE1    .   S
FILE2   .    S

[edit  ][new ed][compil][disk  ]
```

Continue by selecting the files beginning with **FIL** (the **F** has already been typed).

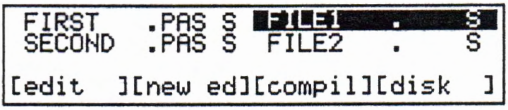
IL



```
FILE1   .    S  FILE2    .   S


[edit  ][new ed][compil][disk  ]
```

When MENU key is pressed, all files will be displayed again, with the cursor bar on the last selected file.

MENU



```
FIRST   .PAS S FILE1   .    S
SECOND  .PAS S FILE2   .    S
[edit  ][new ed][compil][disk  ]
```

Using the EXE key

The selected file is operated on by using the function keys. In addition, the EXE and S EXE keys can be used for some additional functions. The action of these keys depends on the file type of the selected file.

When the file is a sequential file (normal text file with file type **S**), a press on the EXE key will start the edit mode on that file. If S EXE is pressed, the file will be run as a batch file (further explained in chapter 8).

For an executable file (file type **E**), the debugger is normally called when the EXE key is pressed. The debug mode is more thoroughly described in chapter 13. If debug information has been deselected (by the compiler directive {$D–}), the debugger is not called but the program is run directly. Also, if keys S EXE is pressed, the program is run immediately, regardless of any debug information.

If the selected file has any other file type, no operation is carried out when the EXE key is pressed.

[edit] Edit Function

A sequential file can be looked at and changed by this function key. The editor is called when the `[edit]` key is pressed. The functions of the editor are described in chapter 9.

This key is dimmed when the file type is not of sequential type. Such a file cannot be edited directly from the **MENU** mode.

[new ed] New edit Function

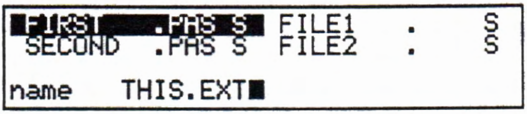
A new sequential file is created by the `[new ed]` function key. This function key will ask for the wanted name of the new file. If that file already exists, the cursor bar will be placed on that file, and the **MENU** mode is continued (the editor will not be called). Otherwise, the file is created and the editor called using the new file.

Example:

The new file **THIS.EXT** is entered as a new file.

`[new ed]`
THIS.EXT

EXE



```
FIRST    .PAS S  FILE1    :    S
SECOND   .PAS S  FILE2    :    S

name   THIS.EXT■
```

[compil] Compile Function

The selected sequential file is compiled by pressing this function key. Chapter 12 describes how the compiler works, and the Pascal language is described in part 3 of the manual.

This function key is dimmed if the file type is not sequential.

[disk] Disk Function

The **MENU** mode can also be used with the files on a floppy disk in the interface unit MD-100. This key is pressed to start the disk menu mode. The functions of this mode are described on page 53.

The `[disk]` function key is dimmed if no MD-100 option unit is connected to the computer.

[name] Rename Function

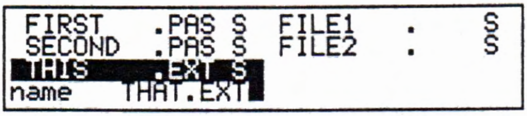
This function key is used to rename a file name. The new file name is entered on the function line. The old file name will be prompted, and can be changed as described in **Input Line Edit** on page 36. The cursor bar will then point to the renamed file.

Example:

Rename the file THIS.EXT to THAT.EXT.

`[name]`
THAT.EXT

EXE



```
FIRST    .PAS S  FILE1    :    S
SECOND   .PAS S  FILE2    :    S
THIS     .EXT S
name   THAT.EXT■
```

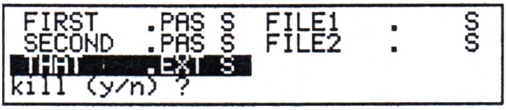
[kill] File Delete Function

Files no longer of use can be deleted by the `[kill]` function key. For safety, the computer will ask for acceptance before deleting the file. At that question, press *Y* or EXE to delete the file. Any other key will abort the delete process.

Example:

Delete the file THAT.EXT.

`[kill]`



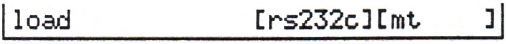
```
FIRST    .PAS S  FILE1    .    S
SECOND   .PAS S  FILE2    :    S
THAT     .EXT S
kill (y/n) ?
```

Y

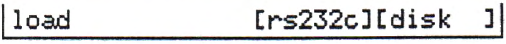
[load] Load Function

By using this key, files can be read from RS-232C, cassette, or disk. The possible devices will be shown on the function line, and the corresponding function key is pressed to specify the device.

`[load]`



```
load               [rs232c][mt    ]
```



```
load               [rs232c][disk  ]
```

If RS-232C is specified, the current communication protocol settings are displayed and can be changed if needed (see page 39). When the EXE key is pressed, reading of data will begin. Data will be read either until a read error occurs, or the ON BRK key is pressed. The cursor bar will then be placed on the new file.

If cassette or disk is specified, the name of the file to copy is entered on the function line. The name of the new file will be the same as the source file name, and the cursor will be placed on this new file. If a file already exists having the same name as the new file, that file will be deleted.

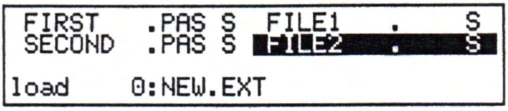
This function key is dimmed if no option unit is connected to the PB-2000.

Example:

Load the file **NEW.EXT** from disk.

`[load] [disk]`
NEW.EXT

EXE

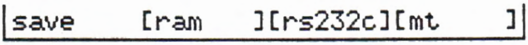


```
FIRST    .PAS S  FILE1    .    S
SECOND   .PAS S  FILE2    .    S

load     0:NEW.EXT
```

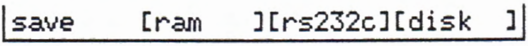
[save] Save Function

Files are saved (copied) to a device by using this function key. The RS-232C, cassette, disk, and RAM devices can be used. If an interface unit is currently connected, the possible destinations are shown on the function line. If no unit is connected, the RAM device is automatically selected.

`[save]`



```
save     [ram    ][rs232c][mt     ]
```



```
save     [ram    ][rs232c][disk   ]
```

For the RS-232C device, the current communication protocol settings are displayed and can be changed. After pressing the EXE key, the data in the file are copied to the serial interface. The transfer can be interrupted at any time by pressing the ON BRK key.

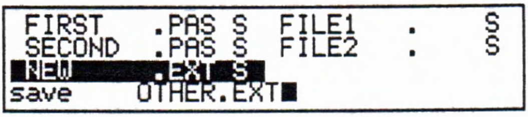
If RAM, cassette, or disk is specified, the destination file name must be entered. The file name of the marked file will be prompted and can be changed. The file will be saved using this file name. For the RAM device, the file name must be changed or an error is generated.

Example:

Save the file **NEW.EXT** under the name **OTHER.EXT** in RAM.

`[save] [ram]`
OTHER.EXT

EXE

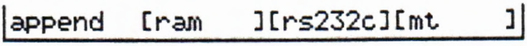


```
FIRST   .PAS S  FILE1   :   S
SECOND  .PAS S  FILE2   :   S
NEW     .EXT S
save    OTHER.EXT■
```

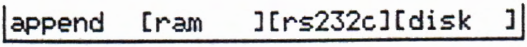
[append] Append Function

A file can be appended to the end of another sequential file using this function key. The file to append can be read from RAM, RS-232C, cassette, and disk, depending on the connected option unit. If no option unit is connected, the RAM device is automatically selected. The read file is appended to the end of the currently selected file.

`[append]`



```
append   [ram    ][rs232c][mt     ]
```



```
append   [ram    ][rs232c][disk   ]
```

If the device is RS-232C, the current communication protocol settings are displayed to be changed, if needed. When the EXE key has been pressed, reading from the serial interface will start. The reading is terminated by a read error or by pressing the ON BRK key.

For the other devices, the name of the file to read is entered on the function line. The corresponding file is then read and appended. The file name of the destination file will not be changed.

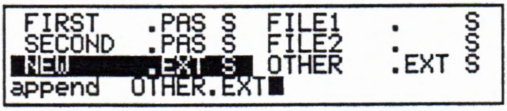
This function key is dimmed if the marked file is not a sequential file.

Example:

Append the file **OTHER.EXT** to file **NEW.EXT**.

`[append] [ram]`
OTHER.EXT

EXE



```
FIRST   .PAS S  FILE1   .    S
SECOND  .PAS S  FILE2   .    S
NEW     .EXT S  OTHER   .EXT S
append  OTHER.EXT█
```

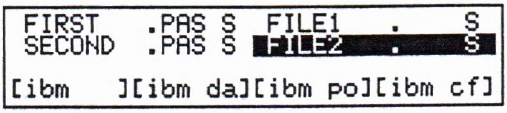
[cnv fr] Convert Function

This key converts a file from an IBM PC character set to the ANSI/ISO character set. The function of this key is described below.

[cnv to] Convert Function

This function key converts a text file to an IBM PC character set.

`[cnv to]`



```
FIRST   .PAS S  FILE1   .    S
SECOND  .PAS S  FILE2   .    S

[ibm   ][ibm da][ibm po][ibm cf]
```

The conversion keys are used to convert the character set used in a file. Conversion is done between an IBM PC character set and the ANSI/ISO character set used by DL-Pascal. The conversion can also be done while transferring the text file on the RS-232C and printer devices.

Four IBM PC character sets can be used for conversion: normal, Denmark/Norway, Portugal, and Canada-French. Some characters cannot be converted, as they are missing in either character set. Such characters are converted to the closest character possible.

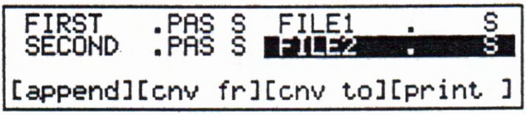
The character sets are displayed on the function line, and the corresponding function key is pressed to select the character set. The file is then immediately converted.

The conversion keys are dimmed if the current file is not a sequential file.

Example:

Convert the file **NEW.EXT** from the Portuguese IBM PC character set.

`[cnv fr] [ibm po]`

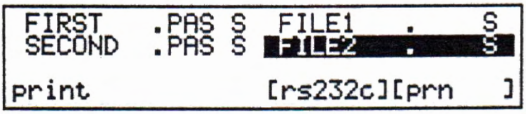


```
FIRST    .PAS S  FILE1    .    S
SECOND   .PAS S  FILE2    .    S
[append][cnv fr][cnv to][print ]
```

[print] Print Function

This function key is used to print a file. The printer can be connected either to the serial (RS-232C) or parallel (Centronics) device. The output device must first be selected.

`[print]`



```
FIRST    .PAS S  FILE1    .    S
SECOND   .PAS S  FILE2    .    S
print            [rs232c][prn  ]
```

For the RS-232C device, the communication protocol is displayed and can be changed. For either device, the conversion code can be added (see page 39). When the EXE key is pressed, the file will be printed.

This key is dimmed if the file type of the selected file does not specify a sequential file.

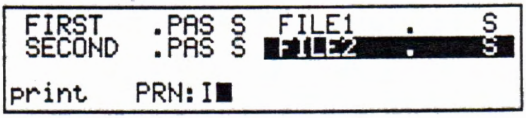
Example:

Print the file **FILE2** to the Centronics printer, with automatic character conversion to the normal IBM character set while printing.

`[print] [prn]`

I

EXE

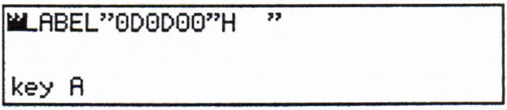


```
FIRST    .PAS S  FILE1    .    S
SECOND   .PAS S  FILE2    .    S
print    PRN:I■
```

[prog.k] Programmable Key Function

The programmable keys can be changed by the `[prog.k]` function key. When this key is pressed, the first function key (S *A*) is displayed.

`[prog.k]`



```
█LABEL"0D0D00"H  "

key A
```

The contents of the programmable key are displayed with normal characters (#$20 to #$7F and #$A0 to #$FF) within quotation marks, and other characters as hexadecimal character pairs (such as 0D in the example above). In the input line, double-character keys (such as the cursor keys) are entered as a null character (#00) followed by the code for that key. These codes are documented in appendix C. All characters may be entered as hexadecimal values, if wanted.

When the EXE key is pressed, the input line is converted and set as the new contents of the programmable key. The next key is then displayed and can be changed.

The key S EXE can also be pressed. This works as the single EXE key, except that the previous programmable key will be displayed instead of the next one.

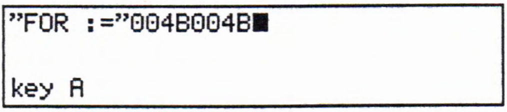
This function is terminated by pressing either of the ON BRK or MENU key.

Example:

Set function key S *A* to write the text *FOR :=* and place the cursor at the colon (two characters back). The code for cursor back is $00,$4B.

`[prog.k]`
"FOR :="004B004B

EXE

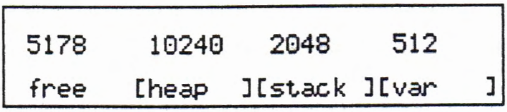


```
"FOR :="004B004B█

key A
```

[memory] Memory Function

This key is used to change the memory allocation of the PB-2000. It will be needed to make room for more variable space or file area.

`[memory]`



```
 5178   10240   2048    512
 free  [heap  ][stack ][var    ]
```

The first value displays the free memory size in the PB-2000. The other values are the current settings of the heap, stack, and **CAL** variable areas. These areas are described in detail in appendix F.

To change a memory area size, the corresponding function key is pressed. The cursor will be displayed, and the new value can be entered.

The entered memory area size will be used when the [EXE] key is pressed. If this setting cannot be used, due to too large a value, an error message is displayed. Otherwise the area sizes are updated in the display to show the new memory setting.

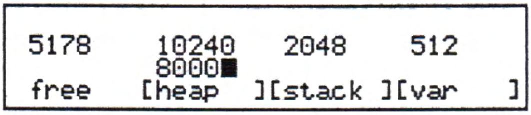
This function is terminated by pressing the [BRK], [EXE], or [MENU] key.

Example:

Set the heap size to 8000 bytes.

`[memory] [heap]`
8000

[EXE]

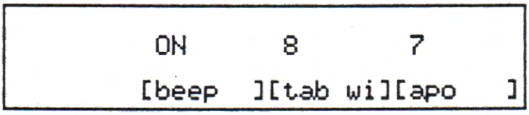


```
 5178     10240    2048     512
          8000█
 free    [heap   ][stack ][var    ]
```

[set] Data Set Function

The `[set]` function key is used to set the key beeper, automatic power off time, and tabulator interval.

`[set]`



```
          ON       8        7

         [beep   ][tab wi][apo    ]
```

The current settings are displayed on the screen. To change the key beeper, press the `[beep]` function key. This key toggles the beeper on or off.

The tabulator interval is used by the tabulation key. The normal value for this is eight, meaning that a tabulation moves the cursor to the nearest larger position on the screen that is a multiple of eight (i.e. position 8, 16 or 24). If the new position would be larger than the screen or window, the cursor moves to the first position of the next line. The tab value is changed by pressing the function key `[tab wi]`, entering the new tab value, and pressing the [EXE] key. The value can be specified in the range 1 to 32.

The auto power off time is in minutes. It is changed by pressing the `[apo]` function key, entering a new value, and pressing the [EXE] key. The value can be in the range 2 to 255 minutes.

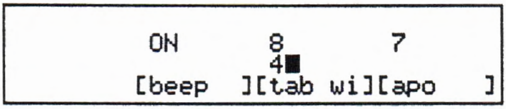
To terminate this function, press the [BRK], [EXE], or [MENU] key.

Example:

Set the tabulator interval to 4 characters.

`[set] [tab wi]`
4

EXE



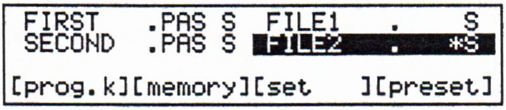
```
        ON       8        7
                 4■
        [beep   ][tab wi][apo    ]
```

[preset] Preset Function

This function key is used to set or remove a file from the **CAL** function keys. Up to four files can be specified for this purpose.

If the marked file has not been set as a preset file, a press on the `[preset]` function key will set the file as a preset file, if there is room for one more preset file.

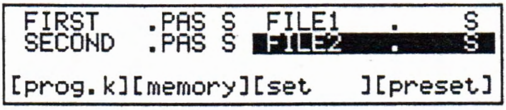
`[preset]`



```
 FIRST   .PAS S  FILE1   .   S
 SECOND  .PAS S  FILE2   .  *S

[prog.k][memory][set   ][preset]
```

A preset file is displayed in the **MENU** mode with an asterisk between the extension and file type. To remove the preset state, just press the `[preset]` function key with the cursor placed on that file. The asterisk will then be removed in the **MENU** display.

`[preset]`



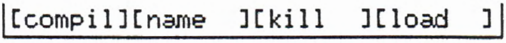
```
 FIRST   .PAS S  FILE1   .   S
 SECOND  .PAS S  FILE2   .   S

[prog.k][memory][set   ][preset]
```

The usage of the function keys in the **CAL** mode is described on page 67.

Disk Menu

The files on a disk in the interface unit MD-100 can also be used in **MENU** mode, by pressing the `[disk]` function key. Files are selected as in the normal **MENU** mode. The EXE key can also be used in the same manner, but the file will first be loaded into normal RAM memory before the function is performed.

In the disk menu, four functions keys are available.



```
[compil][name  ][kill  ][load  ]
```

While being in the disk menu, the disk should not be changed. To replace the disk, first return to the **MENU** mode by pressing the MENU key, replace the disk, and then press the `[disk]` function key. The MENU key is always used to return to the normal **MENU** mode.

If an error occurs in the disk menu, an error message is written and the normal **MENU** mode is selected.

[compil] Disk File Compile Function

This function key is used to compile a Pascal file with the source file on a floppy disk. Compilation in this way allows generation of larger code files than possible with the source file in RAM.

This key is dimmed if the selected file is not a sequential file.

[name] Disk File Rename Function

File names on the disk can be renamed by the `[name]` function key. The new name of the file is then asked for. The function of this key is identical to the normal **MENU** rename function described on page 46.

[kill] Disk File Delete Function

Files can be removed from the disk by this function key. As for the normal **MENU** kill function, the program asks for acceptance before the file is deleted. Answer EXE or *Y* to delete the file.

[load] Disk File Load Function

The `[load]` function key is used to load a disk file into normal RAM memory of the PB-2000. When this key is pressed, loading starts immediately. After loading, the normal **MENU** mode will be selected, with the cursor bar placed on the newly loaded file.

Chapter 7

CAL Mode

In the **CAL** mode, the PB-2000 can be used as a scientific calculator. Previous calculations are saved and can easily be retrieved and recalculated. The heap memory area is used for this purpose. The memory size can be set in the **CAL** mode by using the CLEAR command, and in **MENU** mode by the `[memory]` function key. When there is no more free space in the **CAL** area, the lines at the beginning of the area are automatically removed.

It is also possible to execute commands in the **CAL** mode. This *command line* execution is similar to the entry of commands on a PC. Even batch files are supported. Commands specific to the batch file is described in the next chapter. Some commands need the **CAL** area and will delete the contents of this area, except for the last contents of the display.

Up to four files can be *preset* in the **CAL** mode display. These files can be called by a press on the corresponding function key. The setting of such files can be made in **CAL** mode by the PRESET command, and in **MENU** mode by the `[preset]` function key.

Error messages are displayed on the function line. This message is automatically removed when a key is pressed.

Cursor movement

The cursor can be moved freely within the **CAL** area by using the cursor keys.

[⇨]	Move the cursor one character to the right, or to the first character of the next line.
[⇦]	Move the cursor one character to the left, or to the end of the previous line.
[ANS] [⇨]	Move the cursor as if [⇨] is pressed eight times.
[ANS] [⇦]	Move the cursor as if [⇦] is pressed eight times.
[S] [⇨] LEND	Move the cursor to the end of the current line.
[S] [⇦] LTOP	Move the cursor to the beginning of the current line.

⇩	Move the cursor one screen line down. The position within the line is remembered to allow correct positioning even if an intermediate line is short.
⇧	Move the cursor one screen line up. As with the ⇩ key, the position within the line is remembered.
ANS ⇩	Move the cursor one full screen down (three or four lines depending on any preset files). The line position will be remembered.
ANS ⇧	Move the cursor one full screen up. The position within the line is remembered.
S ⇩	Move the cursor to the end of the **CAL** area.
S ⇧	Move the cursor to the beginning of the **CAL** area.
ON BRK	Move the cursor to the beginning of the next line, without executing the current line.
CLS	Move the cursor to the beginning of the **CAL** area.
S HOME CLS	Clear the **CAL** area and move the cursor to the beginning of it.
ANS TAB	Toggle between the ANSI/ISO character set and the japanese character set.
ETC	If preset files has been set, this key toggles the function line, displaying the files, on and off.
MENU	Move to the **MENU** mode. The **CAL** area is deleted when this key is pressed. This mode is described in chapter 6.
S MEMO IN MEMO	Call the **MEMO** input mode. This mode is described in chapter 10.
MEMO	Search for **MEMO** data, using the characters typed before this key is pressed. This is described in detail in chapter 10.
o NEW ALL	Initialize the computer. When this key is pressed, all memory contents of the computer are initialized, so all files will be deleted!

Data entry

At any time, characters can be entered into the **CAL** area just by typing them. The line length can be up to 255 characters; longer lines will give an error message when the EXE key is pressed. A tabulation can also be inserted by pressing the TAB key. If the insert mode is set (indicated by a box cursor), the typed character is inserted in the line, and the remainder of the line pushed forward. If replace mode is set (indicated by a line cursor), the character will replace the old one at the cursor. Insert/replace mode is toggled between by the INS key.

There are also some editing keys available in the **CAL** mode.

BS	Delete the character to the left of the cursor, or the new-line character separating the lines.
S LDEL BS	Delete the remainder of the current line, including the character at the cursor.
DEL	Delete the character at the cursor, and move the remainder of the line one position to the left.
S DEL	Delete the entire line where the cursor is located.
ANS DEL	Delete the remainder of the **CAL** area, from the cursor onwards.
INS	Toggle between insert and replace mode. The shape of the cursor will change to indicate the new mode.
S INS	Insert the next key pressed as a character, without decoding the key. This is useful to insert special characters like backspace. An empty line can be inserted by pressing S INS ANS EXE (ANS EXE is the actual new-line character).
ANS SPC	Insert the result of the last calculation at the cursor location.
EXE	Execute the current line as an expression or command.
IN	Copy the contents of the current line to the function memory buffer. The cursor will then be placed at the beginning of the next line.
OUT	Insert the current contents of the function memory buffer at the cursor position.
CALC	Delete the remainder of the **CAL** area and start calculation of the function memory. This is further described in chapter 11.

Normal calculations

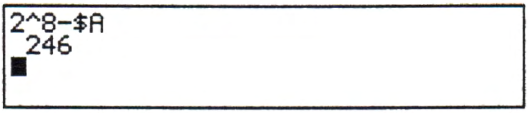
Expressions can be entered in the **CAL** mode using the normal arithmetic operators. As arguments to the operators, real numbers, numeric variables, and mathematical functions can be used.

+	Addition
−	Subtraction
*	Multiplication
/	Division
^	Power

Both normal numbers and hexadecimal numbers can be entered. Hexadecimal numbers must be preceded by a dollar sign.

Example:

2^8–$A EXE



```
2^8-$A
 246
■
```

Variables

Numeric variables are named by using a sequence of letters and digits. The first character in the name must be a letter. The name must not be the same as a mathematical function name.

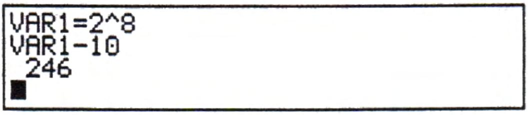
A numeric variable is assigned a value by using the equal sign. Such variables can later be used in the expressions. The contents of the variable is set in the numeric variable area. The size of the area can be changed in **MENU** mode by the `[memory]` function key, and in **CAL** mode by the CLEAR command. All variables will retain their value until the CLEAR command is used. This command will clear the numeric variable area.

If too many numeric variables are assigned a value, filling the numeric variable area, an error message will be written. In this case, use the CLEAR command to clear the area, and (if needed) make the area larger.

When assigning a value to a variable, no result is written on the display.

Example:

VAR1=2^8 EXE
VAR1–10 EXE



```
VAR1=2^8
VAR1-10
 246
■
```

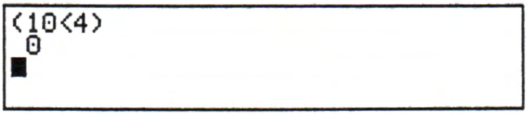
Two expressions can be compared to each other by using the compare operators. The equal sign is also allowed as a compare operator. To separate the two functions of the equal sign, the entire comparison should be enclosed within parentheses.

= < > <= <> >=

If the comparison is false, zero is returned as the result. If the comparison is true, the value one is returned.

Example:

(10<4) EXE



```
(10<4)
 0
■
```

Mathematical functions

There are a number of mathematical functions built into the **CAL** mode.

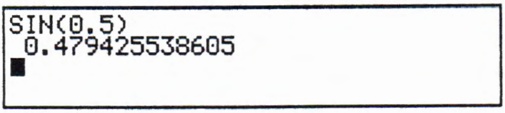
Trigonometric functions

These functions are normally calculated in radians. By using the ANGLE command, the angle unit for calculation can be changed to degrees or grads (see page 62).

`SIN (x)`	Calculate sine of the argument (sin x).
`COS (x)`	Calculate cosine of the argument (cos x).
`TAN (x)`	Calculate tangent of the argument (tan x).
`ARCSIN (x)` `ASN (x)`	Calculate arcsine of the argument ($\sin^{-1} x$).
`ARCCOS (x)` `ACS (x)`	Calculate arccos of the argument ($\cos^{-1} x$).
`ARCTAN (x)` `ATN (x)`	Calculate arctangent of the argument ($\tan^{-1} x$).
`SINH (x)`	Calculate hyperbolic sine of the argument (sinh x).
`COSH (x)`	Calculate hyperbolic cosine of the argument (cosh x).
`TANH (x)`	Calculate hyperbolic tangent of the argument (tanh x).
`ARCSINH (x)`	Calculate hyperbolic arcsine of the argument ($\sinh^{-1} x$).
`ARCCOSH (x)`	Calculate hyperbolic arccosine of the argument ($\cosh^{-1} x$).
`ARCTANH (x)`	Calculate hyperbolic arctangent of the argument ($\tanh^{-1} x$).

Example:

SIN(0.5) EXE



```
SIN(0.5)
 0.479425538605
█
```

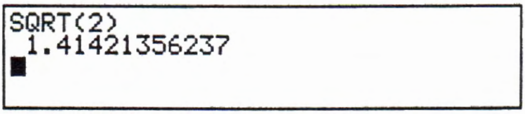
Other One-argument functions

`ABS (x)`	Calculate the absolute value of the argument.
`CEIL (x)`	Round the argument up (return the smallest integer larger than or equal to the argument).

`EXP (x)`	Return natural exponent of the argument (e^x).
`FLOOR (x)`	Round the argument down (return the largest integer smaller than or equal to the argument).
`FRAC (x)`	Return the fractional part of the argument.
`INT (x)`	Return the integer part of the argument (round down if the argument is positive, and round up if the argument is negative).
`LN (x)`	Calculate the natural logarithm of the argument ($\log_e x$).
`LOG (x)`	Calculate the common logarithm of the argument ($\log_{10} x$).
`SQR (x)` **`SQRT (x)`**	Calculate the square root of the argument ($\sqrt{x}$).

Example:

SQRT(2) EXE



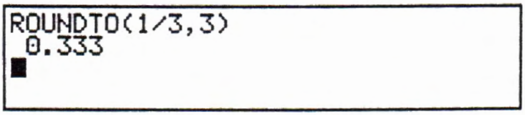
```
SQRT(2)
 1.41421356237
■
```

Two-argument function

`ROUNDTO (x,deci)`	Round x to the specified number of decimals. If the number of decimals is smaller than zero, rounding takes place to the left of the decimal point.

Example:

ROUNDTO(1/3,3) EXE



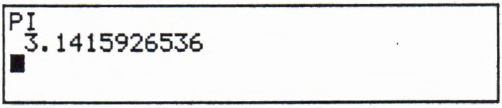
```
ROUNDTO(1/3,3)
 0.333
■
```

No-argument functions

`ERR`	Return the return code (error value) from the last command. This is described on page 71.
`KEY`	Return the key pressed when waiting at PAUSE. This command is described on page 74.
`PI`	Return the (rounded) value of pi used in the PB-2000.
`RND`	Calculate the next random number from the random number generator.

Example:

PI EXE



```
PI
 3.1415926536
■
```

Commands

The **CAL** mode can be used as a command line interface to the computer. This is especially important when batch files are written. Batch files can make it easier to handle groups of files, such as a large programming project.

Each parameter to the command must be separated by at least one space. There must also be a space between the command and the first parameter. If this is not done, a syntax error will be the result. A parameter or file name that has to include at least one space can be surrounded by double quotes. All characters included within the quotes are then treated as one parameter (without the quotes).

Some commands use the area needed by the **CAL** mode. These commands will delete the current area when executed. After execution, the resulting display is copied to the **CAL** area. It is noted at the command description if the command uses the **CAL** area.

There are also some commands that allow *redirection* of the input and output of the command. This can be used to take the input from another place than the keyboard (e.g. input of the same text a number of times to a program), and to send the output to a file or printer. A space must prefix the redirection symbols, and the symbol must be directly followed by a file specifier. The following symbols are allowed as redirection symbols:

<	input redirection from a file
>	output redirection to a file
>>	output redirection, append to a file

The commands allowing for redirection are noted at their description.

Example:

```
DIR >LIST
```

This example will list all files to the destination file **LIST**, instead of the display.

ANGLE

Usage: `ANGLE ang`

Description: Set the angle type for the trigonometric functions. The value ang must be 0 for degrees, 1 for radians, and 2 for grads.

Example:

```
ANGLE 0
```

APPEND

Usage: `APPEND destName sourceName`

Description: This command appends the file sourceName at the end of destName. Any device is allowed for both file specifiers, even the display and/or keyboard.

The CAL area is deleted by this command. The ERR variable will be set to the error code, or, if no error occurs, the variable is set to zero.

Example: Append the file ADD.PAS to the end of TOTAL.PAS:

```
APPEND TOTAL.PAS ADD.PAS
```

BEEP

Usage:

```
BEEP
BEEP ON
BEEP OFF
```

Description: To make a sound, this command is used without any parameter. If the text ON or OFF is used as a parameter, the key beeper is set on or off, respectively.

CLEAR

Usage:

```
CLEAR
CLEAR heapSize
CLEAR heapSize,stackSize
CLEAR heapSize,stackSize,numVarSize
```

Description: The CLEAR command is used to change the memory allocation. If called without any parameters, only the numeric variable area is cleared, but the **CAL** area is intact. For all other uses of this command, the **CAL** area is cleared.

If the numeric variable area size is changed, the variable area is also cleared.

The heap and/or stack size can be omitted if another memory size parameter is needed. The commas must be entered, though.

The use of the different areas is described in appendix F.

The ERR variable is set by this command. If no error is found, this variable is set to zero.

Example:

```
CLEAR 10000
CLEAR ,1000,500
```

The first example sets the heap size to 10000 bytes.

The second example sets the stack size to 1000 bytes and the numeric variable area to 500 bytes. The heap size will not be changed in this example.

CLS

Usage: **`CLS`**

Description: Clear the display and place the cursor at the upper left corner. This command also clears the **CAL** area.

COMP

Usage: **`COMP fileName`**

Description: This command starts compilation on the specified file. The file to compile can be located either in RAM or on disk. If the file is in RAM and a compilation error occurs, the editor is automatically invoked. If the file is on disk, a compilation error will just writes the error code, file name, and line number of the error.

The **CAL** area is cleared by this command. The command also sets the ERR variable to the error code, or sets it to zero if no error occurs.

Example: To compile the file **TEST.PAS** in **CAL** mode:

```
COMP TEST.PAS
```

CONVFR, CONVTO

Usage: **`CONVFR type,fileName`**

Description: Convert a file from (CONVFR) or to (CONVTO) an IBM PC character set. The character set is specified by the type expression, and can be one of the following values: 1:IBM normal, 2:IBM Denmark/Norway, 3:IBM Portugal, 4:IBM Canada-French. The character set used in the PB-2000 is the ANSI/ISO character set.

The characters are converted to the nearest possible one in the other character set. Not all characters can be converted to the equal character, as some characters will be missing in the other character set.

The ERR variable will be set by these commands. If no error is set by the conversion, the variable is set to zero.

Example: The file **READ.TXT** has been read from an IBM PC without any conversion. To convert the file from this character set, write:

```
CONVFR 1,READ
```

COPY

Usage: **`COPY sourceFile destFile`**

Description: This command is used to copy a file to a new file, or to another device. It has the same function as the **MENU** function keys `[load]`, `[save]`, and `[print]`.

If either of the file name, extension, or type of the destFile specifier is omitted, that part is used from the sourceFile specifier. Also, wild card characters can be used in destFile. In this case, the corresponding characters from the sourceFile will be used instead. An empty extension can be specified by entering just the period, without any actual extension characters.

This command both clears the **CAL** area, and sets the ERR variable to the error code of the operation. If no error occurs, the value zero is set in the variable.

Example: To copy the file **TEST.TXT** in RAM to the same name on disk, write:

```
COPY TEST.TXT 0:
```

As no file name or extension is specified for the destination file, the name and extension of the source file will be used. Also, the file type will not be changed.

DEBUG

Usage:

```
DEBUG
DEBUG fileName param1 ...
```

Description: Start the debugger. If no file name is entered, the raw window mode is used (see page 102). In this mode, neither the code nor the data window can be used.

When a code file name is entered, the debugger is called using this file. The extension of the code file does not have to be entered. The debugger mode is described in detail in chapter 13.

The **CAL** area is cleared by this command. Also, the ERR variable is set to the error code returned by the program.

This command can be used with redirection. The redirection will only affect the input and output of the program being debugged, not the debugger itself.

Example: The file **TEST.EXE** is to be called in debugger mode, with the text **Testing** as the first parameter:

```
DEBUG TEST Testing
```

DEFKEY

Usage:

```
DEFKEY key,text
```

Description: This command defines a programmable key. The keys are numbered from 0 to 19, as can be seen on page 36. S *A* has key number zero and S SPC has key number 19.

The new text of the key is entered with normal characters within double quotes. Special and control characters are entered outside of the quotes as hexadecimal numbers (the EXE key as *0D*). Double-character keys (such as the cursor keys) are entered with the first number as *00* and the actual code as the second number. Illegal characters encountered are just skipped.

Example: Define the key S : (key number 10) to the text **CASIO**, and then move the cursor to the end of the line (code $00,$4F as can be seen in appendix C):

```
DEFKEY 10,"CASIO"004F
```

DEL

Usage: `DEL fileName`

Description: Delete the first file matching the file specification given. The file can be located either in RAM or on disk.

This command sets the ERR variable to the error code of the operation, or to zero if no error occurs.

Example: Delete the first file with extension **EXE**:

```
DEL *.EXE
```

To delete the remaining files, move the cursor up one line and press EXE. Repeat this until an error occurs. This can also be written as a batch file, and is demonstrated on page 73.

DO

Usage: `DO fileName param1 ...`

Description: Batch files are started by this command. The file can be located either in RAM or on disk. The full file specifier must be given, including the extension.

This command will clear the **CAL** area. The ERR variable will not be changed by the command itself, but the value will be transported from the last command within the batch file that sets the variable.

Example: To call the batch file **DEL.BAT** with the first parameter *.EXE, write:

```
DO DEL.BAT *.EXE
```

DIR

Usage: `DIR fileName`

Description: List all files matching the fileName specifier. These files can be located either in RAM or on disk.

This command clears the **CAL** area. It also sets the ERR variable to the error code, or to zero if no error occurs.

Redirection is possible with this command. As the display only can show four lines, redirection to the printer or a file is often used with the DIR command.

Example: List all pascal file (extension **PAS**) on the printer:

```
DIR *.PAS >PRN:
```

EDIT

Usage: **`EDIT fileName`**

Description: This command calls the editor with the file name entered. The file must be located in RAM or on disk. If the file is on disk, it is automatically loaded into RAM before the editor is called.

The variable ERR is set to the error code if the file cannot be edited. Otherwise, it is set to the value zero.

Example: The file **SECOND.PAS** (on disk) is to be loaded to RAM and edited:

```
EDIT 0:SECOND.PAS
```

FORMAT

Usage: **`FORMAT`**

Description: This command is used to initialize a disk for usage by the PB-2000. It is needed before the disk can be used, and is normally done directly after purchasing the disks. If the disk has already been used, the FORMAT command will clear all contents of the disk. Therefore this command should be used with care.

PRESET

Usage:

`PRESET`

`PRESET fileName`

Description: The **CAL** mode preset files are set by this command. If no file name is entered, all currently preset files are deselected. Otherwise, the selected file is set as a preset file if not already set, or deselected if it is currently set as a preset file.

The preset files are displayed in **CAL** mode on the function line at the bottom of the display. The first six characters of the file name are displayed. To use the preset file, press the corresponding function key. This will call the file, as if the file was marked in the **MENU** mode, and the EXE key was pressed.

Example: Deselect all preset files, and set only the file **TEST.EXE** as a preset file:

```
PRESET
PRESET TEST.EXE
```

REN

Usage: **`REN fromName toName`**

Description: This command is used to rename a file to another file name. The file can be located either in RAM or on disk. The device of the new name must be the same as used by the old name, and does not have to be entered. The file type cannot be changed by this command. If either the file name or extension is omitted of the new name, that part is used from the old name. Also, if wild card characters are used in the new name, characters at those positions in the old name are used.

The ERR variable is set to the error code of this command, or to zero if no error occurs.

Example: Rename the file **TEST.TXT** to just **TEST** (no extension):

```
REN TEST.TXT TEST.
```

The period is needed to signal an empty extension. If this period is not entered, the extension of the old name is used.

RUN

Usage: **`RUN fileName param1 ...`**

Description: To execute a program, this command is used. The extension of the code file is not needed, just the code file name. The file can be located either in RAM or on disk. If the file is on disk, it is automatically loaded into RAM before execution.

This command will clear the **CAL** area. The ERR variable is set to the error code returned by the program.

Redirection can be used with a program. If not changed by the program (e.g. the Crt unit), the normal Read and Write procedures will use the redirected files.

Example: The program **TEST.EXE** is to be executed, with the first parameter set to **Testing**:

```
RUN TEST Testing
```

VERIFY

Usage: **`VERIFY fileName`**

Description: Verify that the file fileName can be read without errors. The command will just read the file, without placing the read data anywhere. It is useful to check that a disk or cassette file has no read errors.

The CAL area is deleted by this command. The ERR variable will be set to the error code, or, if no error occurs, the variable is set to zero.

Example: Verify that the first cassette file can be read without errors:

```
VERIFY CAS:
```

Chapter 8

Batch Files

Batch files are used to enter a sequence of commands to be executed at a later time. In addition to the commands and expressions described in the previous chapter, some additional commands can be used in a batch file. These commands are used to control execution of the batch file.

One example of a batch file use is saving compiled files on a disk. If there are a number of files for a program (e.g. the program uses some units), it takes time and key presses to save them manually. Instead, a batch file can be written, that copies the interesting files to the disk. This single batch file can then be called every time copying of those files is needed.

The batch file can be called either from the **MENU** mode by selecting the batch file and pressing S EXE, or by using the DO command in the **CAL** mode.

Batch files can be stopped at any point by pressing the ON BRK key. The cursor will move to the first character of the next display line, and normal **CAL** mode will be selected.

Each line in the batch file is echoed to the display when it has been read. This echoing can be suppressed by adding an at-sign (@) to the beginning of each line to be suppressed. For such lines, nothing will be written and the cursor position will not change.

Error messages are always written to the display after the line where the error occurred. This is done even if the echoing of the batch line was suppressed by the at-sign. Errors will also set the ERR variable. This variable can be used in expressions to test the result of certain commands, and then continue execution at another point in the batch file.

Label lines can also be inserted in the batch files. The labels are used by the GOTO command to continue execution at another batch file line. Label lines start with a colon (:) followed by the label name. These lines are not echoed to the display, and must not be preceded by any at-sign. Lower and upper case letters in labels are treated as distinct letters.

The *parameters* used when calling the batch file can be inserted into each batch line by using the percent sign (%) directly followed by a digit. The percent and digit will be replaced by the parameter corresponding to the digit. Parameter 0 is the name of the batch file, and parameter 1 is the first actual parameter. If the **MENU** mode was used to call the batch file, no parameter will be set (not even parameter 0).

Example:

This batch file (named **CP.BAT**) is used to copy all files of a unit. The files will be copied to disk, and if an error occurs the copying is terminated.

```
COPY %1.INT 0:
@IF ERR THEN GOTO end
COPY %1.PAS 0:
@IF ERR THEN GOTO end
COPY %1.UNI 0:
@IF ERR THEN GOTO end
COPY %1.DLE 0:
:end
```

The lines that copies the files will be displayed with the actual file name, so the current file being copied can be seen. The lines testing for any error are not echoed. If an error occurs, batch file execution goes to the label end and stops, as the batch file ends at that label.

The batch file is called from the **CAL** mode with the first parameter being the name of the unit. If the benchmark unit used in chapter 4 is to be copied, the batch file is called with:

```
DO CP.BAT BENCHTEX
```

ECHO, ECHOX

Usage: **`ECHO text`**

Description: The ECHO commands are used to write text on the display from the batch file. Normally it is used in lines suppressed by the at-sign. The difference between ECHO and ECHOX is that after writing the text, ECHO places the cursor on the first position of the next line. ECHOX will not change the cursor position, but keep it directly after the written text.

These commands will clear the **CAL** area. It can also be used with redirection, to write text to a file.

Example:

```
@ECHO First line
@ECHOX Next line
@ECHO , continued
```

The last ECHO text will be written on the same line as the ECHOX text, directly following it. The result on the screen will be:

```
First line
Next line, continued
```

GOTO

Usage: **`GOTO label`**

Description: Continue execution at the specified label. If the label is not found, execution of the batch file stops.

Example: The following batch file will increment variable **X** by one and write the new value to the display. This is repeated until the ON BRK key is pressed.

```
@X=0
:cont
@X=X+1
@X
@GOTO cont
```

IF

Usage: **`IF expression THEN command`**

Description: The expression of the IF command will be evaluated, and if the result is non-zero, the command following the THEN keyword is executed. If the result of the expression is zero, nothing is executed.

The ERR variable will be set only as the result of the executed command.

Example: This batch file will delete all **.EXE** files, until an error occurs.

```
:nextDel
@DEL *.EXE
@IF ERR=0 THEN GOTO nextDel
```

IFEXIST

Usage: **`IFEXIST fileName THEN command`**

Description: This command is used to execute another command only if the file fileName really exists. If it doesn't, nothing is executed.

The ERR variable will be set only by the executed command, not by the test itself.

Example: A file will only be copied if it exists. If it does not exist, nothing will be executed, and no error message will be written.

```
@IFEXIST myfile THEN COPY myfile 0:
```

PAUSE

Usage: **PAUSE**

Description: This command waits for a key to be pressed. The ECHO or ECHOX command is normally used to write a help text before this command. The key pressed can be read by the KEY function. This function can be used in expressions to continue execution depending on the key pressed. For normal keys, the function returns the character table position, as can be seen in appendix D. For double-character keys (such as cursor keys), the value returned is the second code (see appendix C) times 256.

Example: The batch file will write a message, and then wait for a key to be pressed.

```
@ECHOX Change disk and press a key.
@PAUSE
@ECHO
```

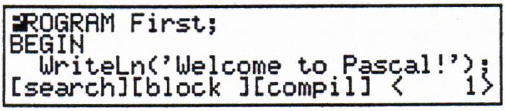
Chapter 9

Editor

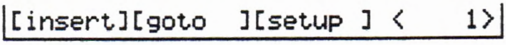
To enter program and sequential data files, the editor is used. This editor is a full-screen editor, capable of moving and deleting blocks, and to search and replace text. Some features of the editor are user-selectable, such as line wrap-around, automatic indentation, and full-display modes. These features are changed by the `[setup]` function key.

The editor is called from the **MENU** mode by using the `[edit]` and `[new ed]` function keys, and from **CAL** mode by the EDIT command.

There are six function keys usable in the editor. They are displayed on the function line at the bottom of the display.



```
PROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
[search][block ][compil] <    1>
```



```
[insert][goto  ][setup ] <    1>
```

The menus are switched by pressing the ETC. key. The first menu can always be selected by pressing S ETC.. If full-display mode has been selected, the ETC. key switches to full-display mode after the second menu has been displayed. One more press on ETC. will select the first menu.

To the right of the menu choices, the current *line number* is displayed. This number increments for each new-line character found up to the current cursor position. A line in the file occupies more than one line on the display if it has more than 31 characters in it.

Error messages are displayed on the function line. The message is automatically removed when a key is pressed.

The **O RESET** button should not be pressed while being in the editor. Since the editor makes as much room as possible for the file being edited, a press on **O RESET** will make the edited file use all available memory space in the PB-2000. If the button has to be pressed, the edited file can be corrected, though. Start the editor on the file being edited when the button was pressed, move the cursor after the last character to be included in the file, and press ANS DEL. This will delete the remainder of the file. Return to the **CAL** or **MENU** area, and the occupied memory will be free.

Cursor movement

The cursor can be moved within the file being edited by using the cursor keys.

⇨	Move the cursor one character to the right, or to the first character of the next line.
⇦	Move the cursor one character to the left, or to the end of the previous line.
ANS ⇨	Move the cursor as if ⇨ is pressed eight times.
ANS ⇦	Move the cursor as if ⇦ is pressed eight times.
S L.END ⇨	Move the cursor to the end of the current line.
S L.TOP ⇦	Move the cursor to the beginning of the current line.
⇩	Move the cursor one screen line down. The position within the line is remembered to allow correct positioning even if an intermediate line is short.
⇧	Move the cursor one screen line up. As with the ⇩ key, the position in the line is remembered.
ANS ⇩	Move the cursor one full screen down (three or four lines depending on whether the menu line is visible or not). The line position will be remembered.
ANS ⇧	Move the cursor one full screen up. The position within the line is remembered.
S ⇩	Move the cursor to the end of the edited file.
S ⇧	Move the cursor to the beginning of the edited file.
ON BRK	Move the cursor to the beginning of the next line.
CLS	Move the cursor to the beginning of the edited file.
ANS TAB	Toggle between the ANSI/ISO character set and the japanese character set.
MENU	Exit the editor and select the **MENU** mode. The cursor bar will be placed on the edited file, even if the editor was called from the **CAL** mode.
CAL	Exit the editor and select the **CAL** mode. If the editor was called from the **CAL** mode, the **CAL** area will be intact.
S MEMO IN MEMO	Call the **MEMO** input mode. This mode is described in chapter 10.

Data entry

Characters can be entered into the edited file at any time, just by typing them. If the insert mode is set (indicated by a box cursor), the typed character is inserted in the line, and the remainder of the line pushed forward. If replace mode is set (indicated by a line cursor), the character will replace the old one at the cursor. Insert/replace mode is toggled between by the [INS] key.

A tabulation can be inserted by pressing the [TAB] key, and a new line is created by pressing the [EXE] key.

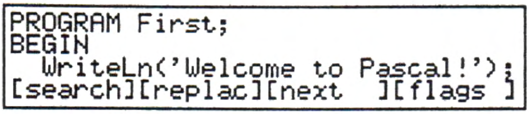
There are also some editing keys available in the editor.

[BS]	Delete the character to the left of the cursor, or the new-line character separating the lines.
[S] [LDEL BS]	Delete the remainder of the current line, including the character at the cursor.
[DEL]	Delete the character at the cursor, and move the remainder of the line one position to the left.
[S] [DEL]	Delete the entire line where the cursor is located.
[ANS] [DEL]	Delete the remainder of the edited file, from the cursor onwards. The contents is moved to the undo buffer, and can be retrieved as described on page 81.
[INS]	Toggle between insert and replace mode. The cursor shape will change to indicate the new mode.
[S] [INS]	Insert the next key pressed as a character, without decoding the key. This can be used to insert special characters like backspace into the file.
[IN]	Copy the contents of the current line to the function memory buffer. The cursor will be placed at the beginning of the next line.
[OUT]	Insert the current contents of the function memory buffer at the cursor position.

[search] Search Function

The `[search]` function key is used to search for a text, or to search and replace a text with another text. If a new setting of the search flags is needed, the flags must be changed before the search or search-and-replace function is called.

`[search]`



```
PROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
[search][replac][next  ][flags ]
```

[search] Search

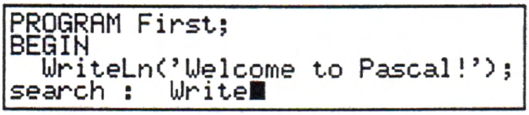
From the search menu, the `[search]` function key is used to search for a text in the file. When this key is pressed, the search text is asked for. The previous search text is prompted, and can be changed (see page 36). Up to 10 characters can be entered as the search text.

When the EXE key is pressed, the text is searched according to the current search flags set.

Example:

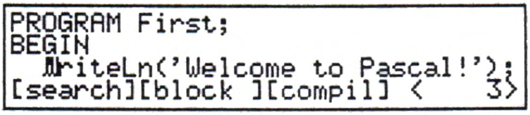
Search for the first occurrence of the text **Write**.

`[search] [search]`
Write



```
PROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
search :  Write█
```

EXE



```
PROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
[search][block ][compil] <     3>
```

[replac] Search and Replace

This function key is used to search for a text and replace it by another text. The replacement text can be empty, if the text searched for is to be deleted.

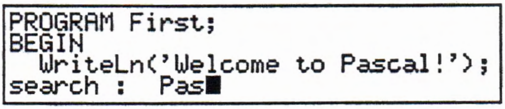
First the search text must be entered. The previous search text is prompted and can be changed. When the EXE key is pressed, the replace text is asked for, with the previous replace text prompted.

After the EXE key is pressed, the search-and-replace is started according to the current search flags set.

Example:

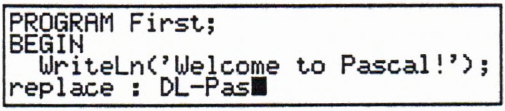
Replace the first text beginning with **Pas** to **DL-Pas** instead.

`[search] [replac]`
Pas



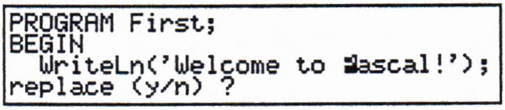
```
PROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
search :  Pas■
```

EXE
DL-Pas



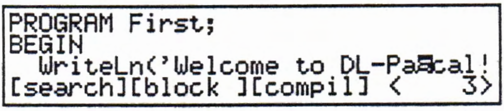
```
PROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
replace : DL-Pas■
```

EXE



```
PROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
replace (y/n) ?
```

Y



```
PROGRAM First;
BEGIN
  WriteLn('Welcome to DL-Pascal!
[search][block ][compil] <    3>
```

[next] Repeat Last Search/Replace

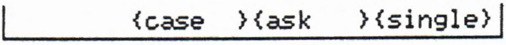
To continue the last search or search-and-replace operation, this key can be pressed instead of moving through the normal questions. The last search/replace operation is immediately repeated when this key is pressed.

[flags] Set Search Flags

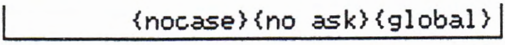
For both the search and search-and-replace operations, some flags can be set to enhance the search function. These flags are for case-insensitivity, replace-without-ask, and global-replace flags. The flags must be changed before the search/replace function is called.

For each flag, there are two states. Either of the states will be displayed in the screen.

`[search] [flags]`



```
        {case  }{ask   }{single}
```



```
        {nocase}{no ask}{global}
```

Press the corresponding function key to change the state of the flag. Use the BRK or EXE key to set the new flags and return to the edit mode.

{case} {nocase} Case Insensitive Flag

This flag specifies the case sensitivity when searching. When set in `{case}` state, the search will only be made for an identical string, with lower and upper case letters as distinct letters. In `{nocase}` state, a letter will match another letter regardless of their casing.

{ask} {no ask} Replace Without Ask Flag

In replace operation, this flag specifies whether a replace acceptance should be asked for, at each found search position in the file, or not. In `{ask}` state, a question is asked when a possible replace position is found. Answer with *Y* or EXE for replace, and another key to skip replacing. The `{no ask}` state is used when no acceptance is needed. In this case, the text found in the file will be replaced without asking.

In any case, the ON BRK key can always be used to stop the replace operation, even if `{no ask}` state is set.

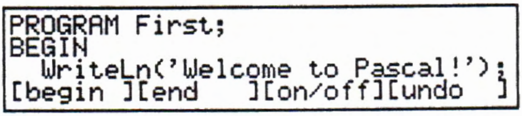
{single} {global} Global Replace Flag

This flag is used in replace operation to set the length of the replace. In `{single}` state, only the next occurrence of the search text will be used for the replacement. To continue the replace operation, the `[next]` function key must be used. The `{global}` state is used to continue the replace operation until the end of the file is reached, or the ON BRK key is pressed.

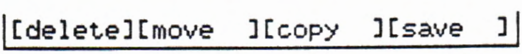
[block] Block Function

This function key handle the *block marking and moving* functions. There are two menu lines to select the needed function. The menu lines are switched between by the ETC key.

`[block]`



```
PROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
[begin ][end    ][on/off][undo  ]
```



```
[delete][move  ][copy  ][save  ]
```

The block function is used in the following way: First of all the block must be marked. A new block is marked by the function keys `[begin]` and `[end]`. The block used the last time can be retrieved by the `[on/off]` function key. The marked block is displayed using reversed characters.

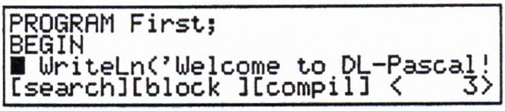
When a block has been marked, the actual block operation can be used. If the block is to be copied or moved, the cursor must be positioned at the destination of the block before the operation is called.

The block will be hidden after a block operation, but the block start and end positions will be updated to the position of the inserted or deleted text.

Example:

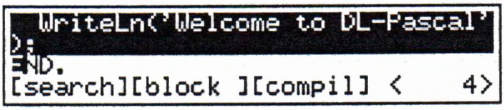
Mark the line that writes the message in the program (starting with WriteLn).

S ⇧
⇩ ⇩
[block] [begin]



```
PROGRAM First;
BEGIN
█ WriteLn('Welcome to DL-Pascal!
[search][block ][compil] <    3>
```

⇩ ⇩
[block] [end]



```
  WriteLn('Welcome to DL-Pascal'
);
END.
[search][block ][compil] <    4>
```

[begin] Set Beginning of Block

This function key is used to set the beginning of the block. The cursor must be placed at the first character in the block before this key is pressed. A new block beginning can be set at any time, even if a block is currently visible.

[end] Set End of Block

The end of the block is set by this function key. For proper position of the end mark, the cursor must be placed after the last character of the block (i.e. at the first character following the block). The end position can be set at any time, regardless of whether the block is visible or not.

[on/off] Unhide/Hide Block

Use this function key to make the last used block visible, or to hide the currently visible block. The block must be visible to allow the use of block operations.

[undo] Undo Deletion

The previously deleted or moved block is copied to the *undo buffer*. By using the `[undo]` key, the contents of the undo buffer is inserted at the current cursor position. It does not have to be the same place as where the block was deleted.

This key is dimmed if no information is currently set in the undo buffer.

[delete] Delete Block

The marked block is deleted from the file with this key. If there is enough room in memory, the block is first copied to the undo buffer. If not, the block is just deleted and cannot be retrieved.

The `[delete]` function key is dimmed if no block is marked, or if the block is not visible.

[move] Move Block

This function key is used to move the marked block to the cursor position. The block is first moved to the undo buffer. If there is not enough room in memory for this, an error is generated, and the move function is stopped.

After copying the block, the marked block is removed from the file. The contents of the undo buffer is then used by this function to insert the saved block in the file at the cursor position.

This function key is dimmed if a block is not marked or the block is hidden.

[copy] Copy Block

The block can also be copied to a new place in the file. This key is used for that purpose. As when moving the block, the block is first copied to the undo buffer. An error is generated if there is not enough memory space for the block to be copied to the undo buffer.

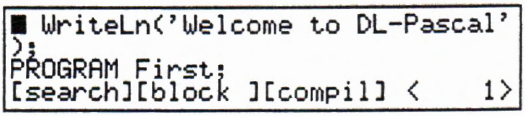
The saved copy of the block is then inserted at the cursor position. If not all of the block can be inserted due to insufficient memory, an error is generated and no text at all is inserted.

If no block is marked or the block is hidden, this function key is dimmed.

Example:

Copy the marked block to the top of the file.

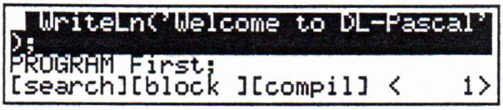
S ⇧
`[block]` ETC.
`[copy]`



```
■ WriteLn('Welcome to DL-Pascal'
);
PROGRAM First;
[search][block ][compil] <    1>
```

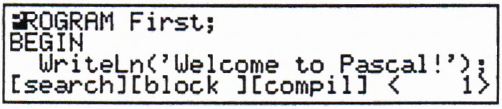
Make the moved block visible, and then delete it.

`[block] [on/off]`



```
 WriteLn('Welcome to DL-Pascal'
);
PROGRAM First;
[search][block ][compil] <     1>
```

`[block]` ETC
`[delete]`

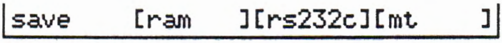


```
■ROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
[search][block ][compil] <     1>
```

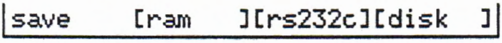
[save] Save Block to File

This function key makes it possible to save the marked block as a new file. If an interface unit is connected to the PB-2000, the possible devices are displayed, and the function key corresponding to the wanted device is pressed. If no unit is connected, the RAM device is automatically selected.

`[save]`



```
save     [ram    ][rs232c][mt     ]
```



```
save     [ram    ][rs232c][disk  ]
```

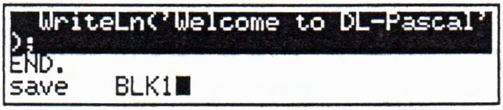
When the device has been selected, the wanted file name is entered. Any existing file with that name will be overwritten.

This function key is dimmed if no block is currently marked, or if the block is not visible.

Example:

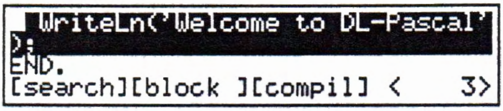
Save the marked block to the new file **BLK1**.

`[block]` ETC
`[save] [ram]`
BLK1



```
 WriteLn('Welcome to DL-Pascal'
);
END.
save    BLK1■
```

EXE



```
 WriteLn('Welcome to DL-Pascal'
);
END.
[search][block ][compil] <     3>
```

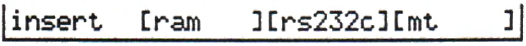
[compil] Compile Function

This function key calls the compiler. The file being edited is used as the source file to the compiler. If no errors are discovered during compilation, the **CAL** or **MENU** mode is automatically selected (depending on which mode the editor was called from). The compiler is described in chapter 12.

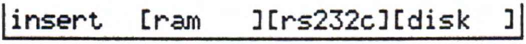
[insert] Insert File Function

Other files can be inserted into the edited file by using this function key. The file will be inserted at the cursor position. If an interface unit is connected, the possible devices are displayed for selection. If no unit is connected, the RAM device is automatically selected.

`[insert]`



```
insert  [ram    ][rs232c][mt     ]
```



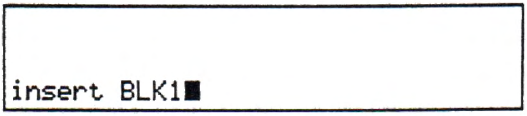
```
insert  [ram    ][rs232c][disk  ]
```

The function key corresponding to the wanted device is pressed, and the file name of the file to insert is entered. That file will then be inserted into the editing file at the current cursor position.

Example:

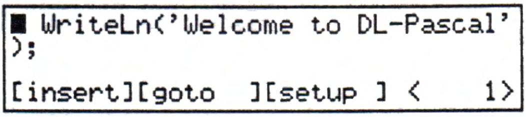
In a new file, insert the previously saved block **BLK1**.

`[insert] [ram]`
BLK1



```
insert BLK1■
```

EXE

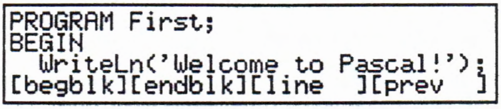


```
■ WriteLn('Welcome to DL-Pascal'
);
[insert][goto   ][setup ] <     1>
```

[goto] Goto Function

By using this function key, fast movement to different parts of the edited file is possible. You can move to the currently marked block, a certain line, and to the previous cursor position.

`[goto]`



```
PROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
[begblk][endblk][line  ][prev  ]
```

[begblk] Goto Beginning of Block

This function key is used to move the cursor position to the beginning of the currently marked block. The key is dimmed if no block is marked, or if the block is hidden.

[endblk] Goto End of Block

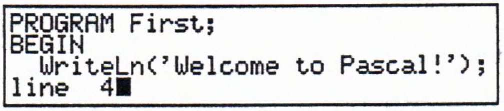
The cursor is moved to the end of the marked block with this function key. If no block is marked, or if the block is hidden, this key will be dimmed.

[line] Goto Line Number

A fast way to move in a large file is to place the cursor at a specific line by using this function key. After pressing the key, the new line number must be entered. When the EXE key is pressed, the cursor is move to that line.

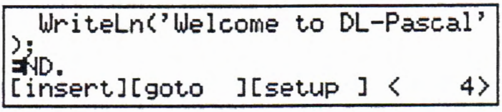
Example:

`[goto] [line]`
4



```
PROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
line  4■
```

EXE



```
  WriteLn('Welcome to DL-Pascal'
);
END.
[insert][goto  ][setup ] <    4>
```

[prev] Goto Previous Cursor Position

After moving the cursor, the cursor can be returned to the previous position by this function key. This is not the position just before the last key was pressed; such a function is rather useless. Instead, all keys are grouped into three groups, selected by the distance of their cursor movement. The previous cursor position is updated only when a key grouped in a higher group is pressed. This allows the cursor to be returned to a position, e.g. after the cursor has been moved to the beginning of the file, down a couple of lines, and a variable declaration has been changed.

The lowest group is the character insert group. This group is used for the character keys, and also the character delete keys such as DEL and BS.

The next group is the short movement group. In this group, short cursor movement keys are placed, such as ⇧ and ⇦. The previous cursor position is only updated if the previous key was in the character insert group.

The highest group is the long movement group. Long cursor movement keys are grouped here, such as S ⇧. The previous cursor position is updated if the previous key was from one of the other groups.

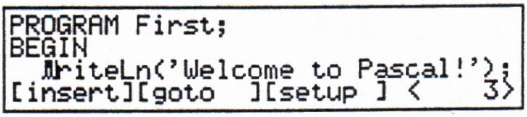
When the `[prev]` function key is pressed, the current and previous cursor positions are swapped, so if this key is pressed again, the cursor is returned to the position before the first press of this key.

Example:

The cursor has been placed at the Write statement (by short movement keys). The cursor is then positioned at the beginning of the file by a long movement key (S ⇧). This will update the cursor position.

S ⇧
⇩ ⇩ ⇨ ⇨

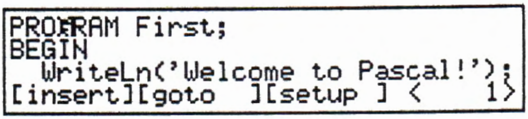
S ⇧



```
PROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
[insert][goto  ][setup ] <    3>
```

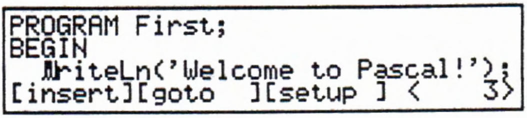
The cursor can then be moved by the cursor keys ⇨ and ⇦, without updating the previous cursor position. Finally when `[goto] [prev]` is pressed, the cursor is returned to the Write statement.

⇨ ⇨ ⇨



```
PROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
[insert][goto  ][setup ] <    1>
```

`[goto] [prev]`



```
PROGRAM First;
BEGIN
  WriteLn('Welcome to Pascal!');
[insert][goto  ][setup ] <    3>
```

[setup] Setup Function

The setup function is used to change the user-installed features of the editor. The user can select between automatic indentation, full-display mode, and line wrap-around. The function line displays the current settings, and to change one setting, the corresponding function key is pressed. One of the two possible choices will be displayed for each function key in the function line.

```
[setup]          {no ind}{menu   }{wrap   }

                 {indent}{nomenu}{nowrap}
```

Press the ON BRK or EXE key to use the new setup mode in the editor.

{no ind} {indent} Automatic Indentation

This flag handles the automatic indentation of lines. Normally (in `{no ind}` state) when the EXE key is pressed, the cursor is placed at the beginning of the next line. In `{indent}` state, the EXE key will also insert the same number of spaces and tabulations as found in the beginning of the current line. This will place the cursor at the same position on the new line, as the first character is positioned on the previous line. This function is very useful when writing Pascal programs, since most lines will be indented the same number of spaces as the previous line.

{menu} {nomenu} Full-display Mode

The normal editor mode is displaying the bottom screen line as a menu line. In `{nomenu}` state, the bottom line can also be used to display a line of the file being edited. In this state, the menu line is displayed by pressing the ETC. key.

{wrap} {nowrap} Line Wrap-around

Since the display often is smaller than a Pascal program line, it is difficult to see the structure of the program in the normal `{wrap}` state. In this state, when a line is longer than 31 character, it continues on the next display line. In `{nowrap}` state, however, a file line always occupy one display line, continuing beyond the right (and/or left) margin of the display. To see the remaining line, the cursor must be moved past these margins. The display will then use horizontal scrolling to display some other part of the line.

Lines must be shorter than 250 characters to be able to see all of it in `{nowrap}` state. Longer lines will appear to be truncated, but no harm is done to such long lines by this mode. To see all of the line, set the flag to the `{wrap}` state.

Chapter 10

MEMO Mode

There is a special memo note pad function in the PB-2000. This is the **MEMO** mode, and is controlled by the MEMO key. Data for the **MEMO** mode is stored in a sequential file named **MEMO**.

The **MEMO** file can be saved on disk as a normal file in the **MENU** (function key `[save]`) or **CAL** mode. Also, the file can be deleted if the data is not needed any more. Deletion is made in the **MENU** (function key `[kill]`) or **CAL** mode.

Input to MEMO file

Input to the **MEMO** file is made by pressing S MEMO IN MEMO. The cursor will be placed at the end of the **MEMO** file. If no **MEMO** file existed, that file will be created before the editor is invoked. Text is entered and edited in the **MEMO** file just like in the normal editor. The only exception is that the `[compil]` function key is dimmed.

Lines can be added between other lines by using the EXE key to add empty lines. Text can then be entered to the new line.

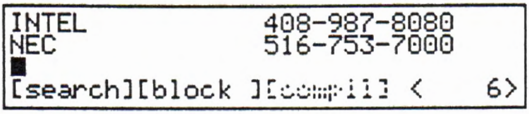
Example:

The following telephone list is to be entered into the **MEMO** function:

CASIO	201-361-5400
COMPAQ	800-231-0900
IBM	914-765-1900
INTEL	408-987-8080
NEC	516-753-7000

Enter the list by pressing S MEMO IN MEMO and then type the text lines, each line separated by the EXE key.

S MEMO IN
CASIO TAB TAB
201-361-5400 EXE
....



```
INTEL              408-987-8080
NEC                516-753-7000
■
[search][block ][compil] <    6>
```

Searching in MEMO file

Data in the **MEMO** file can be searched for by entering the beginning of the search text in **CAL** mode, and then pressing the MEMO key. The first place in the **MEMO** file with a word beginning with the entered text will be displayed. The next place in the file can be located by pressing the MEMO key again. When no more locations are found, the **CAL** mode is returned, with the cursor at the same position as when the MEMO key was pressed.

If the MEMO key is pressed without entering a search text, the cursor will stop at each word in the **MEMO** file.

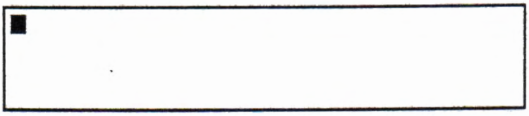
The search in the **MEMO** mode is only made on words separated by spaces or control characters, or on the first word in a line. It will not locate text within words. In addition, lower and upper case letters are treated as different letters. It is easier to search for data if all texts are entered either in all upper or lower case.

While in the **MEMO** search mode, the cursor can be moved around just as in the editor. No text can be entered in this mode, however. To add or edit text, press S MEMO IN first.

Example:

To use the **MEMO** search mode, first press CAL to get into **CAL** mode.

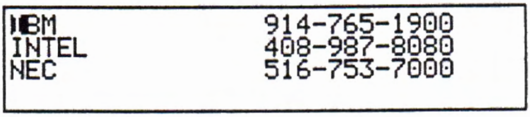
CAL



```
■
```

To search for a company starting with **IB**, press *IB* and then MEMO.

IB MEMO



```
IBM                914-765-1900
INTEL              408-987-8080
NEC                516-753-7000
```

The next company can be located by pressing MEMO once more, or (as in this case) no more companies are found and therefore the **CAL** mode is reselected.

MEMO

```
IB■
```

Chapter 11

Function Memory

The function memory is used for repetitive calculations of one or more expressions involving different values for the variables. The memory can also be used to save text lines in CAL mode and in the editor, as a sort of fast block operation memory.

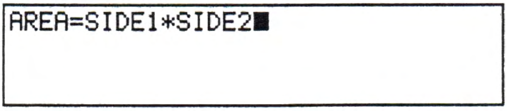
The keys IN and OUT are used to store and retrieve information of the function memory. These keys can be used both in CAL mode and the editor, including the MEMO mode. To calculate using the stored expression, the CALC key is used. This key can only be used in the CAL mode.

Example:

In the CAL mode, store the expression for calculating the area of an rectangle in the function memory.

*AREA=SIDE1 *SIDE2*

IN



```
AREA=SIDE1*SIDE2■
```

Calculate Using Function Memory

The expressions are entered as normal expressions in the CAL mode, and can use all functions available in this mode. No commands can be used when calculating from the function memory. More than one expression can be entered by separating each expression by a colon (:). The IN key is used to store the expression in the function memory. The OUT key can be used to retrieve a store expression, edit it, and later use the IN key to store the edited expression again.

The expression can use a number of variables. When this expression is calculated, the variables in the expression is displayed one by one, and a number or a numeric expression is entered for that variable. The old value for that variable will be kept if no value is specified, but only the EXE key is pressed at the question. The variable will only be prompted one time for all expressions in the function memory, regardless of how many times it is used.

After all variables have been prompted, the expression is calculated and the answer is displayed. By pressing the EXE key again, the following expression is calculated in the same way. If no more expressions has been entered in the function memory, the first one is repeated instead.

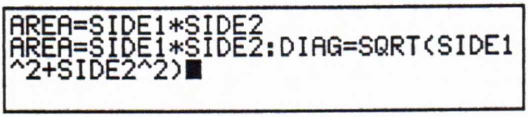
To start calculation, the CALC key is pressed. This key will delete the remainder of the CAL area if the cursor is not at the bottom of this area, to make room for the prompting of variables.

The calculation mode can be stopped by pressing the ON BRK key. The CAL mode will then be reselected.

Example:

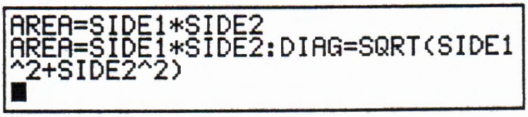
Edit the stored expression, and add an expression to calculate the diagonal of the rectangle.

OUT *:DIAG=SQRT(SIDE1^2+SIDE2^2)*



```
AREA=SIDE1*SIDE2
AREA=SIDE1*SIDE2:DIAG=SQRT(SIDE1
^2+SIDE2^2)█
```

IN

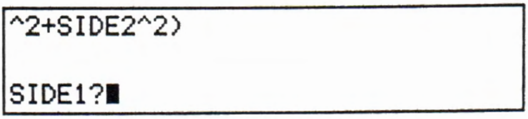


```
AREA=SIDE1*SIDE2
AREA=SIDE1*SIDE2:DIAG=SQRT(SIDE1
^2+SIDE2^2)
█
```

Now let's calculate the area and diagonal of a rectangle with side lengths 6 and 8, using the entered expressions.

Start calculation by pressing the CALC key.

CALC

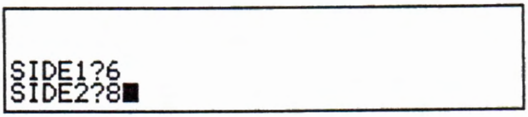


```
^2+SIDE2^2)

SIDE1?█
```

Enter the side lengths of the rectangle.

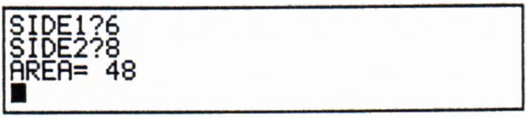
6 EXE *8*

EXE



```

SIDE1?6
SIDE2?8█
```

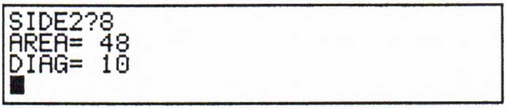
The area of the rectangle is displayed first.



```
SIDE1?6
SIDE2?8
AREA= 48
█
```

Press the EXE key to calculate the next expression.

EXE



```
SIDE2?8
AREA= 48
DIAG= 10
█
```

As both variables had been prompted by the previous expression, they are not prompted again but the result is displayed at once.

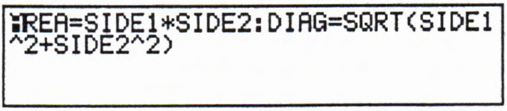
Function Memory and MEMO mode

The **MEMO** mode can be used in conjunction with the function memory. Various expressions can be entered in the **MEMO** file. These expressions can later be entered into the function memory by pressing the IN key while the cursor is placed on the expression. The expression is then calculated by returning to the **CAL** mode and pressing the CALC key.

Example:

The above expressions have been entered into the **MEMO** file. Find the expression by searching for it, and then enter it to the function memory.

AREA MEMO

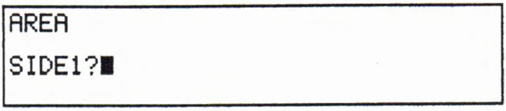


```
AREA=SIDE1*SIDE2:DIAG=SQRT(SIDE1
^2+SIDE2^2)
```

IN

Reselect the **CAL** mode and calculate the expression in the normal way.

CAL CALC



```
AREA

SIDE1?█
```

Chapter 12

Compiler

The Pascal compiler can be called from the **CAL** mode (by the COMP command), the **MENU** mode (by the `[compil]` function key), and directly from the editor. The source file must be a sequential file, and can be located either in RAM or on a floppy disk. Compilation from disk can be used when the total size of the source file and resulting code file is too large for the memory of the PB-2000.

The units imported by the program are first searched in normal memory of the PB-2000. If they are not found, and an MD-100 interface unit is connected, they are searched on the disk instead. If located on the disk, the unit files are automatically loaded into RAM memory.

If the source file is in RAM, a compilation error will directly call the editor and place the cursor at the error position in the program. The error can then be adjusted, and compilation restarted directly from the editor.

If the source file is located on the disk, a compilation error will just write the error message together with the file name and error line number. To correct the error, the file must be loaded into memory. As compilation from disk is normally done because the resulting code file will be large, the corrected program probably have to be saved on disk before compilation can be continued.

After the source file has been compiled successfully, the PB-2000 returns to the previous **CAL** or **MENU** mode, depending on which mode the compiler was called from. If the compiler was called from the **MENU** mode, the cursor will be placed on the created code file.

It is recommended that the source file is saved before execution of the code file. The reason for this is that a badly written program can access any part of the memory, and easily delete or overwrite any files in the PB-2000.

The speed of the compiler is approximately 1000 lines per minute. For a normal program, the size of the code file will be around the size of the source file. If debug information will be used in the code file, the code file will be around 50 percent larger to hold the added debug information. By using the compiler option {$D–} in the beginning of the source file, this debug information can be removed from the code file.

The third part of this manual describes the Pascal language in detail. The identifiers of the standard units are also described there.

During compilation, the ○ RESET button should not be pressed. All remaining memory is allocated to the file being created, and if this button is pressed, no free memory space will be available in the computer. Also, the partially created file will have a bad structure, and cannot be executed successfully. If the ○ RESET button has been pressed, make sure the code file that was being created is deleted immediately.

Chapter 13

Debugger

Often, a compiled program does not work as expected. The *debugger* can then be used to control execution of the program, and look at variables during the program execution. *Break points* can be set in the program, to stop execution at these points in the program. In addition, the BRK key can be used to stop execution when the program is running in a loop. In line input (such as the Read procedure), and when the actual function of the BRK key is needed, the key sequence S BRK must be used instead.

The debugger is automatically called when a program has debug information and is called by the EXE key in **MENU** mode. It can also be called from **CAL** mode by the DEBUG command.

When the debugger is used, all units imported by the program will also use the debugger. If such units has debug information, the debugger will stop at the main (initialization) part of the unit. That unit can then also be debugged. To skip debugging of a unit's main part, just press `[step] [go end]`. The next unit or the main part of the program will then be displayed.

A fully debugged program or unit can be excluded from debugging by using the {$D–} compiler option in the beginning of the implementation part of the unit. No debug information will then be entered in the code file, which also makes the code file smaller.

If you only want to debug certain routines in the program or unit, you can use the {$D–} compiler option at the beginning of the program, and surrounding the parts to be debugged by {$D+} (to set debug information on) and {$D–} (set debug information off). If the main part of the unit has debug information off, this unit will not be stopped at when the program is executed. In order to be able to debug the routines, however, the main part of the actual program being debugged **must** have debug information. If it does not have any information, the program cannot be debugged but will be executed directly.

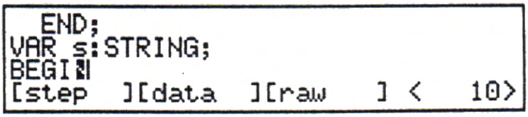
When a program is being debugged, the execution speed of routines in the System and Crt units, as well as some internal functions such as multiplication, are slower than normal. This is the result when execution switches between RAM and ROM. To execute the program at top speed, run it directly by the RUN command in **CAL** mode, or by S EXE in **MENU** mode.

Example:

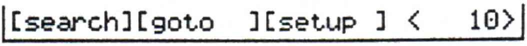
To set debug information only for the procedure Proc below, and in the main part of the program, the debug information compiler option is used.

```
{$D-}
PROGRAM ...
...
{$D+}
PROCEDURE Proc;
...
BEGIN
  ...
END;
{$D-}
...
{$D+}
BEGIN {main}
  ...
END.
```

The first display shown in the debugger is the *code window*. This displays the source code of the program or unit. The cursor can be moved around in the code window in the same way as in the editor. Nothing can be edited in the debugger, however. The function line is also different from the editor.



```
   END;
VAR s:STRING;
BEGIN
[step   ][data   ][raw    ] <    10>
```



```
[search][goto   ][setup ] <    10>
```

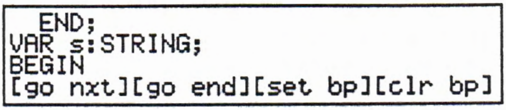
The function key under the line number have a special function in the debugger mode. When pressing this function key, the *output window* of the program (i.e. the screen output from the debugged program) is displayed. Another press on that key (or any other key) will return to the code window.

Examples of some debugging functions are available in chapters 3 and 4 in part one of this manual.

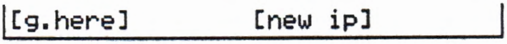
[step] Step Function

The `[step]` function key is used for stepping through the program, and to set and clear break points in programs and units. Up to nine break points can be set in the debugger, in addition to one break point used by the system itself. Two menus are available by this function key, selected by the ETC key.

[step]



```
   END;
VAR s:STRING;
BEGIN
[go nxt][go end][set bp][clr bp]
```



```
[g.here]           [new ip]
```

When execution stops at a break point, that break point is temporarily disabled. It is automatically enabled when another break point is reached. This behavior is needed as the break point replaces the first instruction at the break point. To be able to execute this instruction, that break point cannot be set.

The **O RESET** button should not be used in the debugger, if break points have been set or the ON BRK key has been pressed. As each break point replaces an instruction in the code file, a press on the **O RESET** button will permanently change the code file. If this button has to be pressed (e.g. program loop without being able to break out of it), the code files where the break points were set **must** be deleted and recompiled. If this is not done, execution of that program or unit file will most certainly destroy the contents of the memory in the PB-2000.

Often when the program cannot be stopped by the ON BRK key (possibly together with the S ON BRK key sequence if the computer is waiting for line input), execution has already crashed and destroyed the contents of the memory. In that case, the **O NEW ALL** button also has to be pressed.

[go nxt] Goto Next Statement

Execution to the next statement having debug information in the program is handled by this key. When this key is pressed, a temporary break point is set at the next statement, and execution continues until that statement. Any procedure or function called during execution will not be debugged (unless a break point has been set in that subroutine).

Note that the next statement is the next one as counted from the current statement in the code, not the next one actually executed. The stop position may well be within a conditional statement (e.g. the IF statement), even if that condition is evaluated to False. At such statements, break points should be used instead. This solution has been used due to the processor missing a hardware single stepping facility. It could have been solved in software, but there is not enough memory in the ROM card for this.

This key is dimmed if no debug information can be found for any following statements.

[go end] Goto Program End

This key is used to execute the program to the end, or until the next break point has been found. At unit initialization, this key will execute to the end of the unit's main part, and then set the code window at the main part of the next unit or the program.

[set bp] Set Break Point

Break points are set in the program by using this function key. A break point can be set both in the main program and in the units used by the program. A total of nine break points can be set.

The break point can only be set if there is debug information for the statement at the cursor position. The cursor must be located within the first identifier in the statement to be able to set the break point.

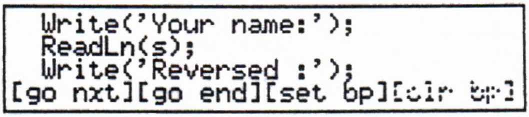
This key is dimmed if no debug information can be found for the statement at the cursor position, or if the cursor is mispositioned in the statement.

Example:

Set a break point three lines down in the program.

⇩ ⇩ ⇩ `[step]`

`[set bp]`



```
   Write('Your name:');
   ReadLn(s);
   Write('Reversed :');
[go nxt][go end][set bp][clr bp]
```

[clr bp] Clear Break Point

The `[clr bp]` function key is used to remove break points no longer needed. When this key is pressed, the first break point is shown in the display. The following keys can then be used to select and remove a break point.

⇩	Move to the next break point.
⇧	Move to the previous break point.
EXE	Remove the break point visible, and return to the code window.
ON BRK	Return to the code window without removing any break points.

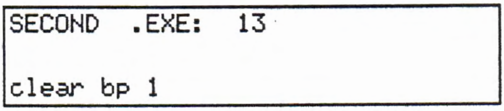
If there are no break points set, this key is dimmed.

Example:

Clear the break point set above.

`[step] [clr bp]`

EXE



```
SECOND  .EXE:   13

clear bp 1
```

[g.here] Goto Cursor Position

Execution can be continued to the current cursor position by using this function key. It can be thought of as setting a temporary break point at the cursor position. The same conditions as for the `[set bp]` function key must be met. If they are not, the `[g.here]` function key is dimmed.

[new ip] Set New Execute Position

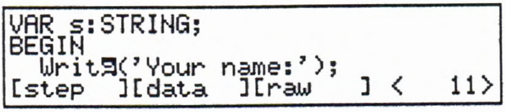
Sometimes a statement in the program has to be skipped or re-executed. This function key allows the current execution position (instruction pointer) to be changed to the current cursor position. The position should only be changed within a subroutine, and only **after** the initial BEGIN keyword has been executed. In other cases, the stack pointer can be pointing to a completely wrong position, resulting in the execution crashing when the subroutine is returned from.

The cursor must be placed using the same conditions as for the `[set bp]` function key. In other cases, this function key is dimmed.

Example:

Skip the first two statements in the program. The BEGIN of the program's main part must first be executed.

`[step] [go nxt]`

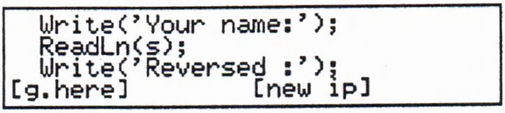


```
VAR s:STRING;
BEGIN
  Write('Your name:');
[step  ][data  ][raw   ] <   11>
```

⇩ ⇩

`[step]` ETC

`[new ip]`

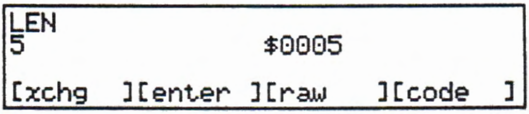


```
  Write('Your name:');
  ReadLn(s);
  Write('Reversed :');
[g.here]         [new ip]
```

[data] Data Window Function

Global variables and local variables in the current subroutine can be looked at with this function key. Also, the contents of the variables can be changed (except for string variables). This display is the *data window*.

`[data]`



```
LEN
5                    $0005
[xchg  ][enter ][raw   ][code  ]
```

If the variable is an array, the first two elements of the array is displayed. Other elements can be shown by using the cursor keys.

The following keys are allowed in the data window.

⇩	Display the next variable.
⇧	Display the previous variable.
⇨	Display the next element in the array.
⇦	Display the previous element in the array.
S LTOP ⇦	Display the first two elements in the array.

The data window will display normal global variables, and local variables and parameters. Variables of integer, real, and string type are displayed in normal way. Enumerated types are displayed as a Byte or Word variable. A Char variable is also displayed as a Byte variable. Short pointers are displayed as a Word variable, and long pointers and subroutine types are displayed as an eight-digit hexadecimal number, with the segment and offset parts combined. All other types are displayed as if they were declared as an ARRAY OF Byte.

[xchg] Exchange Local and Global Display

This function key is used to toggle the display between the local variables and parameters of the current subroutine, and the global variables of the current program or unit.

If the main part of the program is being executed (i.e. no subroutine is being executed), this function key is dimmed.

[enter] Enter New Data

A variable can be assigned a new value by using this function key. All variables except string variables can be assigned a new value. Instead, the raw window is used to change a string variable.

When this key is pressed, the new value is to be entered. For variables of integer type, hexadecimal numbers can be entered by preceding the number by a dollar sign.

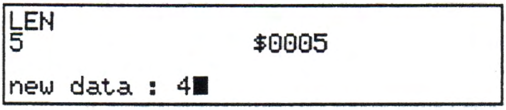
When no variable, or a string variable, is displayed, this key is dimmed.

Example:

Change the contents of the displayed variable `len` to 4 instead.

`[enter] 4`

EXE



```
LEN
5                  $0005

new data : 4■
```

[raw] Set Raw Window

This function key selects the raw window. The current address in the raw window is automatically set to the address of the currently displayed variable. The raw window is described below.

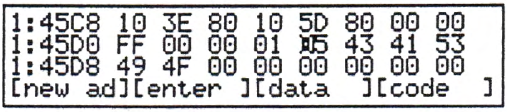
[code] Set Code Window

The code window is selected by this function key. The cursor position will be the same as the last time the code window was displayed.

[raw] Raw Window Function

The *raw window* displays the memory of the PB-2000, in a low-level way. Each line displays eight consecutive memory bytes, for a total of 24 bytes.

`[raw]`



```
1:45C8 10 3E 80 10 5D 80 00 00
1:45D0 FF 00 00 01 05 43 41 53
1:45D8 49 4F 00 00 00 00 00 00
[new ad][enter ][data  ][code  ]
```

The address last displayed in the raw window is used when this mode is reselected. If the raw window was selected from the data window, the address of the last displayed variable is used instead. The address displayed can be changed by the cursor keys.

[⇨]	Move the cursor to the next byte.
[⇦]	Move the cursor to the previous byte.
[⇩]	Move the cursor one line down (eight bytes forward).
[⇧]	Move the cursor one line up (eight bytes backward).

The raw window is the only display possible if the DEBUG command was used from the **CAL** mode, without specifying any parameters.

[new ad] New Display Address

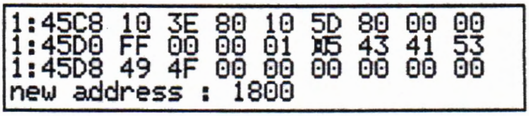
This function key is used to change the cursor address. The new address is entered on the function line as a hexadecimal number. The bank can be changed by entering the bank number followed by a colon and the address within the bank. The normal RAM memory of the PB-2000 is in bank 1.

Example:

Set the displayed address to $1800.

`[new ad]` *1800*

[EXE]



```
1:45C8 10 3E 80 10 5D 80 00 00
1:45D0 FF 00 00 01 05 43 41 53
1:45D8 49 4F 00 00 00 00 00 00
new address : 1800
```

[enter] Enter New Data

New data can be entered in the raw window by using the `[enter]` function key. The byte at the cursor position will be assigned the new data. After this key has been pressed, the new hexadecimal number is entered. Alternatively, a character can be directly entered by preceding the character by a single quote.

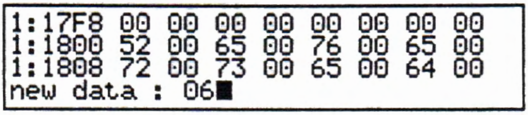
When the [EXE] key is pressed, the address increments automatically and the cursor is placed at the next byte.

Example:

At the previously set address, enter the hexadecimal number $06.

`[enter]` *06*

[EXE]



```
1:17F8 00 00 00 00 00 00 00 00
1:1800 52 00 65 00 76 00 65 00
1:1808 72 00 73 00 65 00 64 00
new data : 06█
```

[data] Set Data Window

The data window is selected by this function key. If DEBUG was called without parameters, this function key is dimmed and cannot be used.

[code] Set Code Window

The code window is selected by this function key. If DEBUG was called without parameters, this function key is dimmed and cannot be used.

[search] Search Function

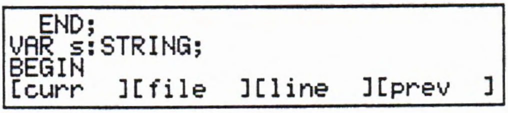
The `[search]` function key has the same function and menu line as in the editor. The only difference is that the `[replac]` function key is dimmed, and cannot be used.

This function key is described on page 77.

[goto] Goto Function

This function key is used to move the cursor position fast within the current file, or to select another unit file used by the current program.

`[goto]`



```
  END;
VAR s:STRING;
BEGIN
[curr   ][file   ][line   ][prev   ]
```

[curr] Goto Current Execution Position

The cursor is placed at the current execution position when this function key is used. This key will return the code window to the current position, even if the cursor is currently positioned in another unit.

[file] Goto Program or Unit File

By using this function key, the cursor can be placed in another unit used by the current program. Global variables in that unit can then be displayed, and break points can be set in the code.

The current program or unit file is displayed when this key is pressed. The code file name is displayed on the first line, and the corresponding source file name on the second line of the display. Another file can be selected by the cursor keys. Only those units having some debug information in them are shown.

⇩	Display the next unit file used by the program.
⇧	Display the previous unit file used by the program.
EXE	Select the currently displayed unit for display in the code window.
ON BRK	Return to the code window without changing the displayed file.

Example:

Select the unit used by the **BenchFor** program.

`[goto] [file]`
⇧

EXE

```
BENCHTEX.DLE
          BENCHTEX.PAS

file
```

[line] Goto Line Number

The `[line]` function key is used in the same way as in the editor, as described on page 85.

[prev] Goto Previous Cursor Position

The cursor position is handled in the same way as in the editor. The previous position can be returned to by this function key. See page 85 for more details about the previous cursor position.

[setup] Setup Function

This function key can also be used in the debugger. The function of it is identical to the function in the editor, and is described on page 86.

Chapter 14

Expanding the PB-2000

The PB-2000 computer can be expanded in different ways: by ROM cards, by a RAM pack, and by two interface units.

ROM Cards OM-51P, OM-53B

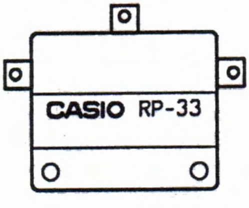
In addition to the Pascal *ROM card,* there are two more ROM cards currently available for the PB-2000. Both these cards are interpreting language cards. The OM-51P is a Prolog interpreter, and the OM-53B is a BASIC interpreter.

Appendix B describes how to change the ROM card. The appendix also describes some precautions regarding the files in the PB-2000 when switching ROM cards.

RAM Pack RP-33

The RAM in the PB-2000 can be expanded to 64 KByte of memory, by using the RAM pack RP-33. The RAM pack is inserted inside of the PB-2000.

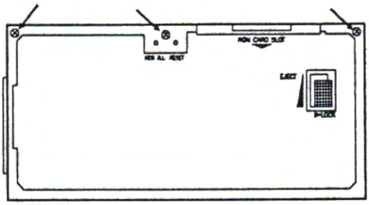
Before inserting the RAM pack, save all important files first. The memory of the PB-2000 have to be initialized after the RAM pack has been inserted, and this will erase all files in the computer.



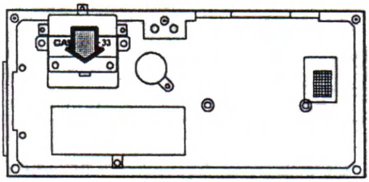
Inserting the RAM pack

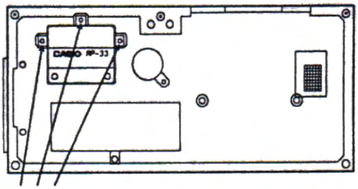
To insert the RAM pack, first make sure that all important files have been saved. Also make sure that you are discharged from static electricity. The easiest way for this is to touch a metal object. Plastic fibers can easily build up static electricity, so you should only wear clothes made of natural material such as cotton while installing the RAM pack.

Make sure the power of the PB-2000 has been switched off. Then place the computer upside down, and unscrew the three screws holding the back of the computer, so the cover can be removed.



Insert the RAM pack into the RAM socket of the PB-2000. The small tongue without a screw hole is first inserted into the corresponding hole in the computer, and then the RAM pack is inserted into the socket. Be sure not to touch the connector on the RAM pack or the socket in the PB-2000. Fingerprints or dust on either connector can make a poor connection and result in improper operation of the computer.





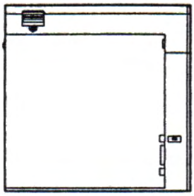
Three screws are included with the RAM pack. Use these screws to secure the RAM pack in the PB-2000. Some pressure on the RAM pack will be needed, for the screws to fit. This is normal, and done this way to assure good contact between the socket of the RAM pack and the computer.

Last, replace the back cover of the computer. Switch power on, press the **○ RESET** button and then the **○ NEW ALL** button. The memory of the computer will now be initialized, and the full 64 KBytes can be used.

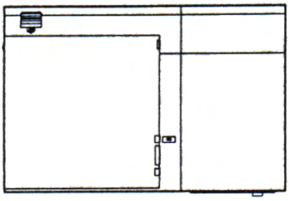
Interface Units FA-7, MD-100

There are two *interface units* for the PB-2000 computer. Both units have a serial and parallel interface. In addition, the FA-7 unit has a cassette interface, while the MD-100 has a floppy disk interface. The interface units can be run either on batteries or on the AD-4175 AC adapter. The batteries are the normal pen-light type (AA or R6). The adapter is connected at the rear of the interface, to the left of the serial connector (when seen from above).

FA-7:

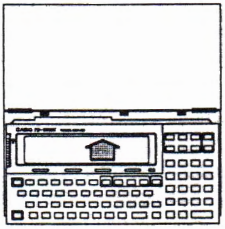


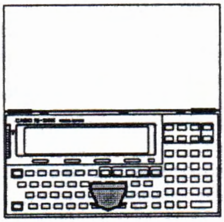
MD-100:



Connect the PB-2000 to the Interface Unit

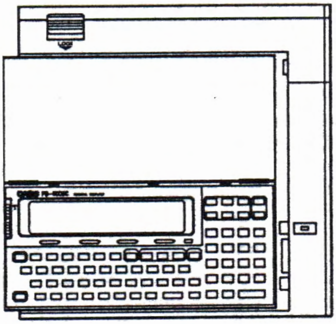
To connect the PB-2000 to the interface, the *hard case* for the PB-2000 must be used. Insert the PB-2000 into the hard case, with the upper side (closest to the display) inserted first, and the lower side gently pressed until the PB-2000 has been fit into the case.

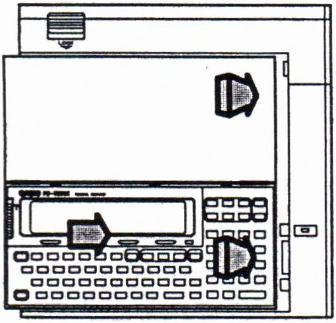


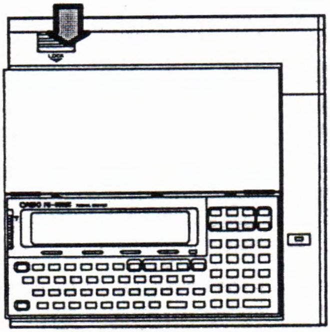


Make sure the power to the PB-2000 is switched off. The PB-2000 controls the power to the interface unit, so the interface unit does not have to be checked for any power switch. Do check that the LOCK switch of the interface unit is in the unlocked state (toward the back of the interface unit).

Place the computer and hard case in the cradle of the interface unit, approximately 1 cm (½") from the connector. With one hand, gently press on the right side (the connector side) of the computer while sliding the computer toward the connector with the other hand. No excessive force should be used to fit the connectors together. There is a small tongue at the upper right corner of the cradle that fits into the hard case. To make the computer fit tightly in the connector, it may be necessary to press that corner of the hard case down.







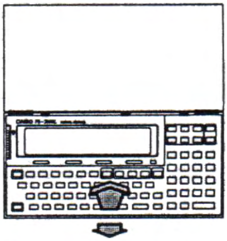
Last, lock the hard case to the interface unit, moving the LOCK switch towards the computer.

Remove the PB-2000 from the Interface Unit

To remove the PB-2000 from the interface unit, first make sure the power of the PB-2000 is turned off and the power indicator of the interface unit is not lit.

Set the LOCK switch of the interface unlocked (toward the back of the interface). The PB-2000 can now be pulled away from the connector, by sliding it to the left. When the connectors are free, the computer can be lifted.

The hard case can now be removed from the PB-2000. To remove it, gently pull the front of the case out, until the front side of the PB-2000 gets free. Lift up the PB-2000 from the hard case, and store the case together with the interface unit.



Serial interface

The serial interface of the interface units is a normal *RS-232C serial interface.* This interface can be connected to other computers, to transfer programs and data. The following pinout is used with the serial interface.

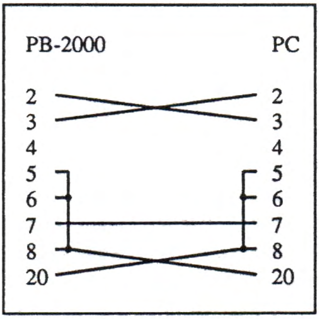
1	Frame ground
2	TxD — output, Transmit data
3	RxD — input, Receive data
4	RTS — output, Request to send
5	CTS — input, Clear to send
6	DSR — input, Data set ready
7	Signal ground
8	CD — input, Carrier detect
20	DTR — output, Data terminal ready

The largest problem with communication over a serial interface is the *handshaking* used by the two connected computers. The same type of handshaking must be used on both computers, otherwise communication may not work.

There are two different types of handshaking: *software* and *hardware.* Software handshaking is normally implemented using the XON/XOFF protocol. The PB-2000 supports such a protocol, and this should be used if the connected computer also uses it. The protocol is specified in the file specifier, as described on page 41.

Hardware handshaking is the handshake type responsible for most problems using a serial interface. By using a cable connected in the way that can be seen in the figure to the right, some of the handshaking lines are deselected. This will normally not affect the communication. The cable will make serial communication work with almost any computer or device.

Before communicating with another computer, you must also be sure that both computers operate on the same speed, number of data and stop bits, and same parity. The file specifier is used to set this information in the PB-2000 (see page 39). The documentation for the other computer or device will describe how to change the settings for that computer. For a PC computer, the `MODE COM1:` command is often used.



There is a special switch on the left side of the interface unit that controls power to the serial interface. This switch is normally set to OFF to preserve the batteries of the interface unit. It is easily forgotten, but **must** be switched ON to make serial communication possible.

Parallel interface

The *Centronics parallel interface* is the most widely used interface for printer connection. Both units support this interface. The pins used by the interface unit is shown below. A normal Centronics-to-Centronics connector cable can be used to connect the printer to the unit.

1	Strobe	19	Gnd
2	Data 1	20	Gnd
3	Data 2	21	Gnd
4	Data 3	22	Gnd
5	Data 4	23	Gnd
6	Data 5	24	Gnd
7	Data 6	25	Gnd
8	Data 7	26	Gnd
9	Data 8	27	Gnd
10	Ack	28	Gnd
11	Busy	29	Gnd
12		30	Gnd
13		31	Init
14		32	Error
15		33	Gnd
16		34	
17		35	
18		36	

There are a large variety of printers available for the PB-2000 computer. All printers having a Centronics interface can be used with the PB-2000. Two small printers are available from CASIO. They are the FP-40 thermal printer, and the FP-100 plotter printer.

Cassette Interface (FA-7)

The FA-7 interface unit has a *cassette interface* for saving programs and data to a cassette recorder. The cable SB-52 should be used to connect the interface and the recorder.

Connect the black plug of the cable to the EAR connector of the FA-7 unit, and to the EAR output of the recorder. This output might be marked LINE OUT instead of EAR.

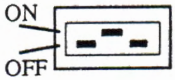
Connect the white plug of the cable to the MIC connector of the FA-7 unit, and to the MIC input of the recorder (this might also be marked LINE IN).

If the cassette recorder have a connector marked REM, then connect the gray plug between that connector and the REM connector of the FA-7. This plug and cable does not have to be connected. The connection is only used to start and stop the tape remotely from the computer.

Cassette Transfer Rate

The transfer rate for saving and loading from the cassette recorder is set on the bottom of the FA-7. One of the setting shown to the right is possible.

bps	*Sw1*	*Sw2*	*Sw3*
300	OFF	ON	OFF
600	ON	ON	OFF
1200	OFF	OFF	ON
2400	ON	OFF	ON



The possible rate of transfer will depend on the cassette recorder used. A faster rate demands more of the recorder than a slower rate. Almost all cassette recorders will be able to use 300 bps transfer rate.

The PHASE switch on the right of the interface unit can be used if you have difficulties reading files from the recorder. Normally this switch should be set to NOR. Some recorders need the switch to be set in REV position.

Disk Interface (MD-100)

The MD-100 interface unit can use standard 3½ inch *floppy disks* to save program and data. Each floppy disk can hold up to 320 KByte of information.

The power indicator of the MD-100 switches to red when the battery power of the interface unit gets weak. The batteries should then be changed as soon as possible, or the AC adapter be connected to the interface. If the MD-100 unit is still run on the weak batteries, the unit will automatically switch off after some time.

Inserting a Floppy Disk

The floppy disk is inserted into the slot at the front of the MD-100 unit, with the label side of the floppy disk upwards. Push the disk into the slot, until it locks in the slot. The disk can then be used by the PB-2000. If the disk is newly purchased, it must be formatted before use. This is made in the **CAL** mode using the FORMAT command (described on page 67).

Removing a Floppy Disk

After using a floppy disk in a program, or in **CAL** or **MENU** mode, it can be removed from the floppy disk drive. Before removing the disk, make sure the access indicator of the disk drive is not lit. The disk, or the data of the disk, might otherwise be destroyed.

Press the eject button of the disk drive (located at the lower right of the drive slot). The floppy disk pops out from the slot, and can then be pulled out of the disk drive.

The floppy disk should be stored dust-free, out of the reach of magnetic fields, and in normal room temperature and humidity, to make sure that the contents of the disk are not affected.

Part 3

Compiler Reference

Chapter 15

Introduction

This reference manual describes the syntax of the DL-Pascal language. This Pascal version follows standard Pascal in large, but due to the restricted memory available in the PB-2000, the compiler has a few changes from the standard. There are also a number of enhancements, to make it easier to write commercial programs.

The differences and enhancements to the standard are shown when the corresponding language construct is described. There is also a separate chapter that compares DL-Pascal with the Pascal standard, and also with the leading two Pascal compilers for personal computers.

EBNF

In order to describe the DL-Pascal *language syntax*, we use a formal language called *EBNF* (Extended Backus Naur Formalism). This formal language allows an easy description of the syntax of a language.

Each *definition line* in EBNF has the following structure:

```
declare:        definition
```

The EBNF language has some *special characters* to describe the definition. These special characters are shown enlarged in this reference. The first character is the *colon* (:), which is used to separate the EBNF identifier being declared from the actual definition of that identifier.

The casing of the identifiers used in EBNF definitions specify the type of identifier. All EBNF identifiers have the first character in lower case (such as sign below), and Pascal characters and reserved words are all upper case (like PROGRAM).

There are a few other special characters. The next is the *vertical bar*:

```
sign:           + | -
```

This means that a sign can either be a + or a –, but not both at the same time (i.e. +– is not allowed as one sign). In the same way, a digit is only allowed to be a single digit in the range 0 through 9:

digit: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The last characters are the *square brackets* and *curly braces*:

intNumber: [sign] digit { digit }

This line defines intNumber (abbreviation for integer number) to possibly have a sign, but it doesn't have to have it (the square brackets denote that the elements within them are optional).

Next, the number must have one digit. This digit can then be followed by any number of digits, or no more digits at all (the curly braces is used when an element may be repeated any number of times, or be totally omitted).

With intNumber declared this way, the following entries are valid:

```
1             71            +8            +9412
-2            -75134887652121
```

The following entries are **not** valid intNumbers:

```
1A            12.34         +             -+9712
```

Chapter 16

Symbols

A DL-Pascal program is seen by the compiler as a sequence of *symbols*. The symbols can be *identifiers*, *numbers*, *strings*, and *single-* and *double-character* symbols. These symbols are described in this chapter.

The symbols must be separated from each other; either implicitly by a character not allowed in that symbol, or explicitly by the use of a separator. Spaces, control characters (such as tabulation and new-line), and comments can be used as separators.

The Pascal language is a *free-format* language. As such, the symbols can be separated by any number of separators. It is fully possible to use only spaces to separate each symbol, and equally possible to only use new-line characters (where each symbol will start on a new line). Both these extreme formats of the program will, however, be difficult to read.

To make the program easier to read, *indentation* of program lines is normally used. When a new indentation *level* is established (e.g. a new procedure, statements within a compound statement or the program body), the lines are indented by some spaces (two spaces are used in this manual). In addition, statements continued from the previous line often has one additional space in the indentation to signal that the statement has been continued. Using indentation in this way makes it much easier to read and maintain the programs. All programs and examples in this manual will use this indentation structure.

Identifier

Identifiers are used to denote constants, types, variables, and subroutines with a descriptive name. The identifier is a sequence of letters, digits, and underline characters. The first character of an identifier must be a letter. Lower case letters are treated as upper case letters. In addition to the normal letters A to Z, the national characters in the range #$80 to #$FF can also be used in an identifier.

singleId: letter { letter | digit | underline }
letter: A | B | C | ... | Z | nationalChar
digit: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The identifier can be of any length, but only the first 31 characters are used.

Some identifiers are reserved by the system, and cannot be redeclared. They are called *reserved words* (or *keywords*). In DL-Pascal they are:

```
ABSOLUTE    ELSE        INLINE      PROCEDURE   TO
AND         END         INTERFACE   PROGRAM     TYPE
ARRAY       FILE        LABEL       RECORD      UNIT
BEGIN       FOR         MOD         REPEAT      UNTIL
CASE        FORWARD     NIL         SET         USES
CONST       FUNCTION    NOT         SHL         VAR
DIV         GOTO        OF          SHR         WHILE
DO          IF          OR          STRING      WITH
DOWNTO      IN          PACKED      THEN        XOR
```

Identifiers can be prefixed by the unit or program name where they were declared. This is useful when two identifiers from different units are used, and the identifiers have the same name. This is further discussed on pages 187 and 155.

id: [singleId .] singleId

Examples:

```
LongIdentifier          Proc3       Next_Call
Unit.variableName
```

When a new identifier is to be used, it is shown in the EBNF definition by the use of newId instead of singleId.

newId: singleId

To build a list of new identifiers, each identifier is separated by a comma. There must be at least one identifier in the list.

newIdList: newId { , newId }

Both lower and upper case characters can be used in an identifier, even though they are read by the compiler as they are all in upper case. By using a different combination of lower and upper case in different types of identifiers, the program is easier to read and understand.

The *casing of identifiers* has changed over the years. The first style was to use lower case, underlined letters for the reserved words. Next style has been to show reserved words in lower case boldface. But in most cases, neither underline nor boldface can actually be entered in a program editor. The program displayed on the screen will therefore not show any difference between the reserved words and other identifiers.

The style adopted in this manual is the "modern" way, used by the successor to Pascal: MODULA-2. In this style, reserved words are all upper case (as in the reserved words table above). Furthermore, constant, type, unit, and subroutine identifiers begin with an upper case letter in each separate word building the identifier (as in LongIdentifier). Label, variable, record field, and enumeration identifiers are also cased this way, except that the very first letter of the identifier is a lower case letter (as in variableName). This casing style will be used in all examples.

Number

The numbers express a value in the Pascal program. There are two types of numbers: integer and real.

number: integer | real

Integer

Integer numbers use a sequence of digits without decimal or exponent specifiers. They are faster than real numbers when used in a program.

Hexadecimal integers (numbers of base 16) can also be entered. They are prefixed by a dollar sign, with the hexadecimal number directly following it.

The resulting integer number must be in the range 0 through 2^{31} (2147483648).

integer: digit { digit } | $ hexDigit { hexDigit }

hexDigit: digit | A | B | C | D | E | F

Examples:

```
0          42576102    $123F
```

Real

Real numbers can have a much larger range of values than integer numbers, and will therefore execute slower than integer numbers. The range is from –9.99E–99 to 9.99E+99. The number must include a decimal point. It can also have an exponent.

The exponent specifies the magnitude of the number (i.e. the number of left or right shifts of the decimal point in the number).

real: digit { digit } . { digit } [E exponent]
exponent: [sign] digit { digit }
sign: + | -

Examples:

```
0.          12.34       7345.E32    8.345E-25
```

String

A string is used to hold text in the Pascal program. String text in the program is enclosed in single quotes. All characters except control characters can be directly entered in the string. A single quote character can be included in the string by double-typing the quote. If a control character is included within the quotes, the compiler gives an error message.

Any character (also control characters) can be included in the string by using the number sign (#) directly followed by the integer value of the character wanted. Such a number character must immediately precede or succeed the quoted string. Any combination of number strings and quoted strings can be put together to form one constant string, as long as there is no separator between them. The character values for different characters are shown in appendix D.

string: subString { subString }
subString: ' characters ' | # integer

Examples:

```
'This is a string'
'Separate'#13#10'lines'
#7'Wake up!'
#$0D#$0A
'A single quote ('') character'
```

Single-character symbol

Some characters have a special function is a Pascal program. These character symbols are:

```
+ - * / . , : ; < = > [ ] ( ) { } ^ @ & %
```

Double-character symbol

Some of the single-character symbols can be put together to form a double-character symbol. In this case, no separator is allowed between the two characters (otherwise they are taken as two separate symbols).

The double-character symbols are:

```
.. <= >= <> := (* *)
```

Comment

Comments can be inserted in the text using two different symbol pairs: Either delimited by the curly braces { and } or the double-character symbols (* and *). The comment must start and end with the same type of delimiters (e.g. a comment cannot start with symbol (* and end with a curly brace }). Nesting of comments using the same delimiter types is not possible.

If the very first character in the comment is a dollar sign, then the comment specifies a *compiler option*. These options are explained in chapter 33.

Comments (except compiler options) are skipped by the compiler, and therefore takes no space in the compiled program.

comment: { characters } | (* characters *)

Examples:

```
{ This is a comment }
(* This is a long comment
   over two lines *)
{$I INCLUDE.FIL }
(*$F-,R-,S-,T-*)
```

Chapter 17

Program

A Pascal program has the following structure:

progDecl: [progHead]
[usesDecl]
declarations
body .

These parts are described in this chapter.

The Pascal program (the source file) will be compiled into an executable file (the code file). The code file will have the same name as the source file, but with extension **EXE** and file type **E**.

Program heading

Normally, a *program heading* begins the program. This heading declares the program identifier (the name of the program), which should be the same as the file name of the program. This identifier can later be used to prefix global identifiers (identifiers declared at the main program level) when the same identifier has been used in a procedure or function.

progHead: PROGRAM newId [(newIdList)] ;

The optional identifier list following the program name is used in standard Pascal to specify the files used by the program. In DL-Pascal, all files used by the program are declared in a more general way. Therefore, this identifier list has no function, and it is just skipped over if it has been entered.

In DL-Pascal, the program heading is optional, if there is no need to declare the program identifier.

Examples:

```
PROGRAM NewProgram;
PROGRAM MyProgram (this, is, skipped);
```

Uses declaration

The next part of program is the *unit import* list. This list specifies all separate compiled units needed by this program.

usesDecl: USES newIdList ;

There is one predefined unit always imported automatically. This is the System unit. In this unit, all system types, variables, procedures and functions are declared. Since the System unit is automatically imported, it should not (in fact cannot) be included in the USES declaration.

If no units are imported, the USES declaration is not needed.

For all identifiers in the USES list, the compiler tries to locate the corresponding interface unit files in the computer, using the first eight characters in the identifier, converted to upper case, as the file name. If a unit file is not found, and a floppy disk is connected, the program tries to load the file from the floppy disk. A compile error is generated if the file still cannot be located, or if there is not enough room in memory to load the unit file.

Examples:

```
USES Crt;
USES MyUnit, NextUnit, LastUnit;
```

Declarations

Before an identifier can be used in a Pascal program, it has to be declared. This is what the *declaration part* of the program is used for.

declarations: { oneDecl }

oneDecl: labelDecl | constDecl | typeDecl | varDecl | procFuncDecl

The declarations can be entered in any order, as long as the identifiers are declared before they are used. This is not allowed in standard Pascal, and that makes it more difficult to write large programs.

The identifiers declared at this level (the main level) are called *global identifiers*, as they are visible to all subroutines in the program.

The syntax and usage of the various declarations are documented in chapters 18 to 22.

Body

The actual program (the *main part* of the program) is entered in the *program body*.

```
body:          BEGIN
                statList
               END
statList:      statement { ; statement }
```

The statements of a program are separated by semicolons. The last statement before the END keyword doesn't have to end with a semicolon, but it is no error if a semicolon is there. The syntax of the statements is documented in chapter 24.

The last character in a Pascal program is a period, to signal the compiler that the end of the program has been reached. It is possible to have more text entered after this period character, but that text will be never be read by the compiler.

Chapter 18

Label Declaration

Labels are used for non-structured jumps between statements. The label must be declared before it is used, and then can only be used within the subroutine where it was declared.

labelDecl: LABEL newIdList ;

The label identifier is in this reference manual denoted labelId.

labelId: singleId

In standard Pascal, only numeric labels are allowed. In DL-Pascal, however, only identifier labels are allowed (no numeric labels). This is one of the few differences from standard Pascal that exist in DL-Pascal. It is easy to convert from numeric labels to identifiers, though. Just add some letters before the number (e.g. L_), and the label has been converted to an identifier.

Labels are only used with the GOTO statement. This statement is very seldom used in a Pascal program. The only actual need is to jump out of a bunch of nested, structured statements due to a termination or error condition. The GOTO statement is normally misused by programmers converting from BASIC or other non-structured languages.

The destination for the jump is specified by preceding the destination statement by the label identifier and a colon. This is further described on page 181.

Examples:

```
LABEL LabelId;
LABEL Label1, NextLabel, EndLabel;
LABEL L_1, L_999;
```

Chapter 19

Constant Declaration

A program can be easier to read and maintain, if often-used constant values are declared as *constant identifiers*. These constant identifiers do not make the compiled code any larger. They are evaluated and used only at compilation time.

constDecl: CONST oneConstDecl { oneConstDecl }

oneConstDecl: newId = constExpr ;

An identifier declared by the CONST keyword is in this reference shown by constId.

constId: id

Standard Pascal only allows constant identifiers to be assigned constant values of integer, real and string types. In DL-Pascal, any expression is allowed that can be evaluated without generating any code. This allows the constant to be of integer type, enumerated type, character, real, string and set types (the types are described in the next chapter).

The possible operators and arguments possible in the expression are described in chapter 23. With set and string types, no operators are allowed in a constant expression, as all expressions with operators to these types will always generate code.

Examples:

```
CONST Lines = 25;
      LinesButLast = Lines-1;
CONST
  CText = 'Constant text';
  HalfSet = [0..127];
  MaxValue = 9.99E99;
```

Chapter 20

Type Declaration

Variables and subroutine parameters used in a Pascal program are assigned a special type in their declaration. Simple types can be used directly at variable declaration, but more complex types are preferably declared separately in the *type declaration.*

typeDecl:	TYPE oneTypeDecl { oneTypeDecl }
oneTypeDecl:	newIdList = typeElement ;
typeElement:	pointerType | stringType | procFuncType | ordinalType | realType | typeId | [PACKED] strucType

The simplest type element is a previously defined type identifier in this program or any imported unit (including the System unit).

typeId: id

A previously declared type can be used to declare a new type. The new type will inherit all properties of the basic type, including type compatibility. The new type is said to be an *alias* of the basic type. Normally this is used to describe the contents of some variables in a better way.

Example:

```
TYPE
  WinNumber = Integer;
```

Structured types can have the reserved word PACKED preceding their declaration. In standard Pascal, this allows the computer to pack as much information as possible into one storage element. It has no effect in DL-Pascal, but is allowed for compatibility reasons.

strucType: recordType | arrayType | setType | fileType

Ordinal Type

The simple types in Pascal are called *ordinal types*. These types can be handled and stored by the computer in an easy way, and therefore are the most effective types to use in a program.

The different ordinal types in Pascal are: integer types, character type, enumerated type, and subranges of these types.

ordinalType: intType | charType | enumType | subrangeType

Integer types

There are five different *integer types* in DL-Pascal. These integer types can hold different ranges of values. The reason for this is that the computer handles smaller integers more efficient that larger integers. By restricting a type to handle just a small number, the variables occupy less space, and the program will also run faster.

intType: typeId

The following integer types are declared in the System unit in DL-Pascal, and their allowed range of values are:

```
Byte        0 .. 255
ShortInt    -128 .. 127
Word        0 .. 65535
Integer     -32768 .. 32767
LongInt     -2147483648 .. 2147483647
```

As you notice, one integer type is actually called Integer. Normally in the text, when integer type is specified, this means one of the five integer types can be used. When only one of the integer types is meant, the special identifier typeface will be used, and the first letter of the identifier will be in upper case.

Character type

The *character type* can hold a character with an ordinal value between 0 and 255. This type is declared in the System unit with name Char. The character corresponding to a character value is listed in appendix D.

charType: typeId

Enumerated type

It is possible in Pascal to declare your own ordinal types. Such types are called *enumerated types*.

enumType: (newIdList)

Each identifier in the entered list corresponds to an *ordinal value* in the enumerated type. The first identifier has the ordinal value zero, the next value one, and so on. Each element identifier in an enumerated type is denoted enumId in this reference.

enumId: id

The enumerated types are strongly type checked, so you cannot mix an enumerated type with other types, including other enumerated types, by mistake. It is possible to mix different types by using special type conversion functions in DL-Pascal.

The Boolean type declared in the System unit is an enumerated type with the following declaration:

```
TYPE Boolean = (False, True);
```

The boolean type is a special enumerated type, as it has some special capabilities not shared by other enumerated types.

Examples:

```
TYPE
  WeekDays = (mon, tue, wed, thu, fri, sat, sun);
  CardDeck = (spade, heart, diamond, club);
```

Subrange type

Sometimes, only a short range of values is needed, where it will be an error if a value is outside of that range. For this purpose, the *subrange type* is used.

subrangeType: constExpr .. constExpr

Both constant expressions must be of the same type, and the value of the second expression must be larger than or equal to the first one. The subrange type will be the smallest type possible based on these expressions.

Ordinal types (integer, character, and enumerated types) can be used as the base type of a subrange type.

When a value is assigned to a variable declared as a subrange type, the compiler will insert code to check that the assigned value is within the allowed range of values. Note that the compiler option {$R+} must be active for this range check code to be inserted.

The most used instance of a subrange type is in declaration of an array type (see the description of the array type below).

Examples:

```
TYPE
   SmallInt = 0..15;
   WorkDays = mon..fri;
   UpperCase = 'A'..'Z'
```

Real Type

Floating-point numbers are handled by the *real* type. The real type is declared in the System unit as the type Real.

realType: typeId

The real type can hold values in the range –9.99E99 to –1.E–99, 0, and 1.E–99 to 9.99E99. The number is stored with 13 digits.

If underflow of a real number occurs, the number will be replaced by value 0.0. Overflow of a real number will generate a run-time error, except if the {$F–} compiler option has been set.

Example:

```
TYPE
   RealType = Real;
```

Pointer Type

Pascal allows a program to keep variables in places not known at compilation time. *Pointers* are then used to store the address to such variables.

pointerType: ^ [^] id

The base type of a pointer can be of any type. Two pointer variables are compatible only if the variables have identical type. When dereferenced, however, the compatibility rules of the base type is used.

In DL-Pascal, there are two types of pointers; *near* (or *short*) and *far* (or *long*). The normal pointer is a near pointer. A near pointer always points in the normal RAM area of the PB-2000 (bank 1), and is specified by a single caret. A far pointer can point to any memory address in the PB-2000, and it's specification includes two carets. Far pointers need more time to calculate the address, and therefore take more time to execute.

One use of pointers is with memory allocation. In this way, complex memory structures can be built without having to specify the memory size when writing the program. All needed memory is allocated during execution. This is described in detail in chapter 26.

There are two system pointer types declared in the System unit: Pointer and LongPtr. They are compatible with any pointer type, and can be used when no type checking of pointers is needed or wanted. Any near pointer is compatible with the Pointer type, and any far pointer with the LongPtr type.

With pointers, there is a relaxation of the need to declare an identifier before the identifier is used. This is often needed with record types and linked lists, where one or more fields need to point to the next entry, and the next entry is of identical type as the current entry. The type being pointed to must be declared later at the same procedure level as the pointer type, or a compiler error is generated.

Examples:

```
TYPE
  NearInteger = ^Integer;
  FarChar = ^^Char;
```

String Type

A *string* in Pascal holds a string of characters. The maximum number of characters depends on the declared range in the type declaration. The absolute maximum is 255 characters.

stringType: STRING **[** [constExpr] **]**

The expression sets the maximum number of characters in the string. If omitted, the string can hold 255 characters. A string type without a specified maximum length is called the *system string* type.

Examples:

```
TYPE
  NameStr = STRING [40];
  FullStr = STRING;
```

Procedure and Function Type

As an extension to standard Pascal, a type can be declared to be a *procedure* or *function* type. This facility is useful when writing a subroutine that needs to call another subroutine without knowing which one it can be at compilation time. One example is a graph-writing program that is used to write graphs to a number of mathematical functions. The actual math function to display is then declared as a function type.

procFuncType: PROCEDURE procHead | FUNCTION funcHead

The style of the procedure and function headings are described on page 151.

Examples:

```
TYPE
  MathFunc = FUNCTION ( x : Real ): Real;
  NormProc = PROCEDURE;
```

Record Type

In the same way as enumerated types makes it possible to declare a simple user type, the *record type* makes it possible to declare a structured user type. The record is used to group a number of variables together. Each variable place in a record is called a *field*. The fields can be of any type, even other record types. There is no time or memory penalty in the compiled program when using record types.

recordType: RECORD
  [recFieldList]
  [recVariant]
  END

recFieldList: recField { ; recField } [;]

recField: newIdList : typeElement

In a record, different fields can be specified to start at the same address. This is known as a *variant part* of the record. The variant part must be the last field in the record. Variant record declaration is useful when there can be different fields in the record, depending on the usage of the record.

All fields used within one record type must have unique names. However, it is possible to have a field that has the same name as another identifier outside of the record, or as another field in another record. It is also possible to have fields of the same name in a record used within another record, as fields in that record.

recVariant: CASE [newId] : ordinalType OF
[recVarPart]

recVarPart: recVarList { ; recVarList } [;]

recVarList: constExpr { , constExpr } : (recFieldList)

The type used at the CASE must be an ordinal type, except the LongInt type. This *tag type* is used to control the grouping of fields in the variant part of the record. Optionally, an identifier can be declared at the same time.

A small difference from standard Pascal exists in the tag type declaration. In DL-Pascal, the colon separating the optional identifier from the tag type must be entered, even if the identifier is not entered. This is not the case in standard Pascal.

Each variant list in the variant part has a constant expression specifying when these fields are to be used. The type of this expression must be of the same type as the tag type. In each list, a variable number of fields can be declared. These fields (within a list) are stored at consecutive addresses. The different variant lists start at the same address.

No error checking is made in the compiler to assure that the correct fields are used with respect to the tag type. This feature can be used to interpret a type in smaller parts than is normally possible. For example, the following declaration makes it possible to extract the offset and segment parts of a far pointer. It should be noted that this type of interpretation is often non-portable (i.e. will not necessarily work on another machine or compiler).

```
TYPE
  FarPtr = RECORD
             CASE :BOOLEAN OF
             FALSE : (p : LongPtr);
             TRUE  : (offset,segment : Word)
           END;
```

The size of the record type is the size of the fixed record fields plus the largest variant field type. The total size of a record type must be larger than zero.

When a record type is declared, it might be needed to declare one or more fields as a pointer to the same type as the one being declared. This is e.g. needed if a linked list is being constructed. In such case, the pointer type can be declared before the record type, even though the record type has not yet been declared. This is a relaxation of the rule that an identifier must be declared before it is used.

Examples:

```
TYPE
  Date = RECORD
           year  : Word;
           month : 1..12;
           day   : 1..31;
         END;
  PersonPtr = ^Person;
  Person = RECORD
             nextPerson : PersonPtr;
             lastName,
             firstName : STRING [20];
             birthDate : Date;
             CASE living : BOOLEAN OF
             TRUE : ( currAddr  : STRING [40];
                      currState : STRING [30] );
             FALSE: ( deathDate : Date );
           END;
```

Array Type

An *array* consists of a fixed number of instances of the array's base type. The base type itself can be an array; there is no limit on an array other than that it must fit in memory.

arrayType: ARRAY [ordinalType { , ordinalType }] OF typeElement

A multi-dimensional array can be declared with the ranges separated by commas. The type declaration

```
ARRAY [0..9,0..24] OF Integer;
```

is the same as the declaration

```
ARRAY [0..9] OF
  ARRAY [0..24] OF Integer;
```

The range is specified by an ordinal type, except a LongInt type. Often this will be a subrange of an ordinal type. The *elements* of the array are numbered from the minimum value to the maximum value. The range 0..9 will make ten elements (numbered from 0 to 9).

Examples:

```
TYPE
  CharUsed = ARRAY [Char] OF BOOLEAN;
  Screen = ARRAY [0..3,0..31] OF RECORD
                                   ch   : Char;
                                   attr : Byte;
                                 END;
```

Set Type

When a variable is needed to hold a number of distinct values from the same type at the same time, the *set* type is used. A set consists of one bit for each possible value in the set's base type.

setType: SET OF ordinalType

The base type of a set can be any ordinal type that has ordinal values in the range 0 to 255. This means that only enumerated types (including Boolean), Byte, Char, and subranges of these types are possible as a set's base type.

In the example, the attributes for the screen is declared. A set is used for this, as more than one attribute can be set at one time. It is also possible that no attribute is set at all (making an *empty set*).

Examples:

```
TYPE
  Attribute = (atLine, atDim, atRev, atR3,
               atR4, atR5, atR6, atFont);
  AttrSet   = SET OF Attribute;
  NumberSet = SET OF 0..9;
```

File Type

To be able to read from and write to files in memory and on a floppy disk, *file* variables are used. Files can also be used to write to other devices in the PB-2000.

fileType: typeId | FILE [OF typeElement]

There are three basic file types in DL-Pascal: *typed file*, *untyped file*, and *text file*. These file types are more thoroughly described in chapter 27.

Typed file

A typed file is declared with an element type. This element type can be any type except a file type.

Examples:

```
TYPE
  NumberFile = FILE OF Integer;
  DateFile   = FILE OF Date;
```

Untyped file

The untyped file has no element type. This type of file is used when file operations on a low level is needed.

Example:

```
TYPE
  UntypedFile = FILE;
```

Text file

A special file type in DL-Pascal is the text file. This file type is declared in the System unit as Text. Text files are operated on in a sequential manner, and can be connected to any device in the computer, such as the screen/keyboard and the printer.

Example:

```
TYPE
  TextFile = Text;
```

Type Compatibility

The Pascal language has strict type checking, in order to catch as many programming errors as possible already at compilation time. Still, the programmer **can** specify that types can be used together with other, different, types by using *type conversion*. The compiler assumes the programmer knows what he (or she) is doing when using type conversion. Type conversion is described on page 169.

There are three levels of compatibility: *type identity*, *assignment compatibility*, and *expression compatibility*.

Type Identity

This is the most restricted level of compatibility. It is used when a variable-declared subroutine parameter is called.

To allow a variable type to be used for this parameter, at least one of the following conditions must be true:

- Types are identical; i.e. the actual type (non-alias and non-subrange basic type) are the same.
- One type is a pointer type, and the other is a system pointer type of the same size.
- Declared parameter is untyped.
- Declared parameter is an open array, and the called parameter is a variable or array of the same base type.
- Both types are strings, and the compiler option {$V–} is set.
- Both types are procedures of the same number of parameters and equal parameter types.
- Both types are functions of the same number of parameters, equal parameter types, and equal return type.

Assignment Compatibility

This level of compatibility is used when assigning a value to a variable, and when a value-declared subroutine parameter is called.

Assignment compatibility exists when at least one of the following conditions is true:

- Types are identical; i.e. the actual type (non-alias and non-subrange basic type) are the same, except for file and text types, and open arrays.
- One type is a pointer type, and the other is a system pointer type.
- Destination type is a string type, and source type is a string or character type.
- Destination type is an integer or real type, and source type is an integer type.
- Both types are procedures of the same number of parameters and equal parameter types.
- Both types are functions of the same number of parameters, equal parameter types, and equal return type.

Expression Compatibility

This type of compatibility is used within expressions. Expression compatibility differ between different operators and different argument types. This is further explained at these operators in chapter 23.

Chapter 21

Variable Declaration

In the *variable declaration* the variables to be used in the program are declared. The variable type can be entered directly in the variable declaration, but usually more complex types are declared separately.

varDecl: VAR oneVarDecl { oneVarDecl }

oneVarDecl: newIdList : typeElement ; |
newId : typeElement ABSOLUTE absVar ; |
newId : typeElement = initVar ;

A variable identifier is shown in this reference by varId.

varId: id

In addition to the standard variable definition, DL-Pascal allows variables to be declared at an absolute memory address or at the same address as another variable. It is also possible to initialize a variable to a certain value at the start of the program or subroutine.

Variables declared at the main level are called *global variables*. The contents of the global variables are cleared before execution starts.

Absolute Variables

A declared variable is normally placed at the address following the last variable. By using the ABSOLUTE keyword, variables can instead be set at the same address as another variable, or at a certain memory address.

absVar: varId | [constExpr :] constExpr

If ABSOLUTE is followed by a variable identifier, the address of the new variable will be the same as that variable.

Example:

The following code will make variables myValue and myAddr be set at the same memory address:

```
VAR
  myValue : Word;
  myAddr  : Pointer ABSOLUTE myValue;
```

Variables can also be set at an absolute memory address in RAM (e.g. at system variables). This is specified by following the keyword ABSOLUTE by a constant expression. Alternatively, both a bank number and an offset expression can be entered (separated by a colon), to set the variable in any part of the PB-2000's memory space.

Examples:

```
VAR
  memEnd    : Word ABSOLUTE $0AB4;
  copyRight : STRING ABSOLUTE 0:$FFE7;
```

Initialized Variables

Variables can be initialized at the start of the program or subroutine main part, by following the variable type with an equal sign and the initial data.

initVar: constExpr | arrayInit | recordInit

A variable of a simple type is assigned the initial value by a constant expression. In addition, arrays and records can also be assigned.

arrayInit: (initVar { , initVar })

recordInit: (recFieldInit { ; recFieldInit })

recFieldInit: singleId : initVar

In the case of an array, the elements of the array are initialized by specifying the value of each element, separated by commas. The whole list must be enclosed within parentheses. If there are more elements in the array than there are initial values, the remainder of the array will be cleared.

Also record variables can be initialized. In this case, each field of the record is assigned a value, by entering the field name followed by a colon and the initial value. Each field assignment is separated by a semicolon. Fields in the record that are not assigned any value are cleared. The order of assignment must be made in increasing offset within the record. This also means that only one variant list should be used for initialization.

Initial values for complex structures may be nested, so e.g. an array of records can be initialized.

Example: Given the following type declarations:

```
TYPE
  Array4  = ARRAY [1..4] OF Integer;
  Record2 = RECORD
              field1, field2 : Byte;
              field3 : Array4;
            END;
```

the following initialized variable declarations are allowed:

```
VAR
  myVar1 : Char = 'A';
  myVar3 : Array4 = (1, 2, 3, 4);
  myVar2 : Record2 = ( field1 : 1; field2 : 2;
                       field3 : (17, 23));
```

In this example, myVar2.field3[3] and [4] will have the value zero, as no initial value has been entered for these record elements.

Chapter 22

Procedures and Functions

A program can use *subroutines* to hold common parts of a program, and also to structure the program. There are two types of subroutines: *procedures* and *functions*. A procedure is just called; it does not return any value. A function will return a value when it returns to the caller, and a call must therefore be part of an assignment or expression.

procFuncDecl: PROCEDURE newId procHead ; subrBody ; |
FUNCTION newId funcHead ; subrBody ;

In the procedure heading, the parameters are entered. Parameters are used to hold variables specific to one call to the subroutine. It is also possible to use a subroutine without any parameters.

procHead: [(declParaList)]

In the function heading, the parameters of the function are entered, together with the function's return type. The declaration of the parameters is made exactly as in the procedure heading.

funcHead: procHead [: declType]

The possible return types of a function are ordinal types, real, and string types. The declaration of a return type is optional only when the function has previously been forward-declared (see page 156).

The return value of the function is set by assigning a value to the function identifier itself. This assignment must be made within the function.

Example:

```
FUNCTION ThisFunc : Integer;
  BEGIN
    ...
    ThisFunc := 1;
  END;
```

The function **must** assign a value before returning, or else any value that happens to be at the temporary return variable area will be returned. Often, this results in program errors that are difficult to trace. The compiler does not detect such errors.

Parameters

The parameters of a subroutine are used to transfer variables and values to the subroutine. They can also be used to return values to the caller of the subroutine.

declParaList:	declPara { ; declPara }
declPara:	valuePara | varPara | untypePara | openPara
declType:	typeId | STRING

There are four different kinds of parameters in a subroutine: *value parameters*, *variable parameters*, *untyped parameters*, and *open array parameters*. Each one has a special function in a subroutine.

Value Parameters

The most used parameter type is the value parameter.

valuePara: newIdList : declType

When the subroutine is called, an expression is entered at the place of a value parameter. The subroutine can use this value, and even modify it if needed, without disturbing any variables used in the calling expression.

For a parameter of real type, both a real and an integer expression can be used. The compiler automatically converts the result of the expression to a real type.

For structured types, the subroutine must copy the contents of the variable to a temporary area on the stack. This can take a little time, and also some stack space. Structured types that are not changed in the subroutine should normally use variable parameters, to avoid this copying.

It is possible to disable the copying of structured types in a subroutine. This is made by using the compiler option {$C–}. Normally, the option flag should be set, which is also the default setting. When reset, no copying to the stack is made, which saves time and heap space. If the flag is reset, be sure to set the flag again at the end of the subroutine, not to disturb any other subroutine's parameter copying.

Examples:

```
PROCEDURE Proc1 ( value : Integer );
FUNCTION Func1 ( name : STRING; len : Byte ): Word;
```

Variable Parameters

A parameter declared as a variable parameter can only be called with a variable, never with an expression. The reason for this is that only the address of the variable is used as the parameter.

varPara: VAR newIdList : declType

Variable parameters are useful when large data structures are needed as a parameter, since only the address to the data is used. The actual data is never copied, as is the case with value parameters.

Parameters of variable type can also be used to return data from a subroutine. As the address to the variable is used, any change to the parameter within the subroutine also affects the variable used when calling the subroutine.

Only variables declared in normal RAM memory, or as near pointers, can be used as a variable parameter.

Examples:

```
PROCEDURE Proc2 ( VAR ret1,ret2 : Integer );
FUNCTION Func2 ( VAR s : STRING ): STRING;
```

Untyped Parameters

When no type checking of the parameter type is needed, untyped parameters can be used.

untypePara: VAR newId [: ARRAY]

Both syntaxes have the same function; the ARRAY keyword can be used if a full parameter declaration is wanted.

As with variable parameters, only the address to the variable entered at the place of the parameter is used when calling the subroutine. A variable of any type can be used at the parameter.

An untyped parameter cannot be used on it's own in a statement or expression. It must first be type converted, using variable type conversion. This is described on page 169.

Example:

The following function scans for the first byte having a zero value, returning a value starting from one, or zero if none is found.

```
FUNCTION ScanZero ( VAR data : ARRAY;
                        size : Word ): Word;
  TYPE ByteArray = ARRAY [1..60000] OF Byte;
  VAR  p : Word;
  BEGIN
    FOR p := 1 TO size DO
     IF ByteArray(data)[p]=0 THEN BEGIN
       ScanZero := p;
       EXIT;
     END;
    ScanZero := 0;
  END;
```

The function can be called with different variables for that parameter.

```
VAR a : ARRAY [1..100] OF Byte;
    s : STRING;
...
  res := ScanZero (a, 100);
  res := ScanZero (s[1], Length(s));
```

Open Array Parameters

An open array parameter is like a variable parameter. It is useful with a subroutine that must be able to handle arrays of a specific base type, but can be called with different number of elements in the array.

openPara: VAR newId : ARRAY OF id

As with variable parameters, only the address to the array is set when calling the subroutine. The actual number of elements in the array, if needed, must be entered as a separate parameter. The first element of the array will have index zero, and the index for the last element will be total number of elements minus one.

When calling a subroutine with an open array parameter, either a variable of the same type as the array's base type, or an array of the base type of the open array, must be entered at the place of the parameter.

Example:

A sort routine for integer values can be written to handle an integer array of any size. The sort algorithm is called *Straight Selection.*

```
PROCEDURE Sort ( VAR arr : ARRAY OF Integer;
                     size : Integer );
  VAR o,i,tst,tstVal : Integer;
  BEGIN
    FOR o := 0 TO size-2 DO BEGIN
      tst := o;  tstVal := arr[o];
      FOR i := o+1 TO size-1 DO
       IF tstVal>arr[i] THEN BEGIN
         tst := i;  tstVal := arr[i];
       END;
      IF tst<>o THEN BEGIN
        arr[tst] := arr[o];
        arr[o] := tstVal;
      END;
    END;
  END;
```

When calling the sort routine, any integer array can be used.

```
VAR a1 : ARRAY [-10..10] OF Integer;
    a2 : ARRAY [0..99] OF Integer;
...
  Sort (a1, 21);
  Sort (a2, currElements);
```

Subroutine Body

In the body of the procedure or function, the declarations and main part are entered.

subrBody: declarations body | FORWARD

The declarations and main part of a subroutine are entered in the same way as the global declarations and main part of the program. All identifiers declared in a subroutine cease to exist when the last END in that subroutine's body has been decoded. Therefore, any identifiers in a subroutine are only visible to that subroutine, and to the subroutines declared within that subroutine. Such identifiers are therefore called *local identifiers.*

When an identifier have been declared in the subroutine using the same name as a global identifier, the global identifier can still be used by prefixing the identifier name with the program name (entered in the program heading) and a period.

Example: The following procedure declaration is given:

```
PROGRAM Ex1;
VAR a : Integer;
PROCEDURE Level1;
  VAR a,b,c : Word;
  PROCEDURE Level2;
    VAR b,c,d : Word;
    PROCEDURE Level3;
      VAR c,d,e : Word;
      BEGIN
        { level 3 }
        Ex1.a := 1;
      END;
    BEGIN
      { level 2 }
    END;
  BEGIN
    { level 1 }
  END;
...
```

In procedure Level3, the variable a from Level1, and b from Level2 are visible, in addition to the local variables c, d, and e. The global variable is still usable by prefixing it with the program name, even though the variable name has been redeclared in Level1.

In procedure Level2, the local variables b, c, and d are visible, and variable a from Level1. Variable e in Level3 is not visible, since it ceased to exist at the END of that procedure body.

Forward-declared Subroutines

The reserved word FORWARD is used to declare the subroutine heading in one place in the program, with the actual subroutine's body later in the program. This is needed when different subroutines want to call each other, as a subroutine can only be called if it has previously been declared.

The complete subroutine heading is entered, directly followed by the FORWARD keyword. This declares the subroutine's parameters. Later when the body of the subroutine is to be entered, the subroutines type (PROCEDURE or FUNCTION), followed by the subroutine's name and a semicolon, are entered. Thereafter, the actual body of the subroutine can be entered.

It is possible to enter the complete subroutine heading when the forward-declared subroutine is redeclared. In this case, the compiler checks that the two subroutine headings are compatible in respect to the number of parameters and the parameters' types. The actual names of the parameters used in the subroutine are those given by the first heading.

Example:

```
FUNCTION Sub1  ( v : Word ): Word; FORWARD;
FUNCTION Div2  ( w : Word ): Word;
  BEGIN
    Div2:=w + Sub1(w DIV 2);
  END;
FUNCTION Sub1;
  BEGIN
    IF v>0 THEN Sub1:=v + Div2(v-1) ELSE Sub1:=0;
  END;
...
```

Since these two functions call each other, the function Sub1 is first declared as a forward function. Next is function Div2 declared, which uses function Sub1. Last, the body of function Sub1 is declared. This function then uses Div2.

Chapter 23

Expressions

In Pascal, *expressions* are formed by combining variables and values with *operators*. Most operators are *binary*; i.e. they connect two arguments. A few operators are *unary*; i.e. they precede one argument and work on that argument only.

expression: simpleExpr [compareOp simpleExpr]
compareOp: < | = | > | <= | >= | <> | IN
simpleExpr: term { addOp term }
addOp: + | – | OR | XOR
term: factor { mulOp factor }
mulOp: * | / | DIV | MOD | AND | SHL | SHR

The operators have different levels of priority. Operators at a higher level are evaluated before a lower level operator:

High unary operators, arguments
mulOp (*,/,DIV,MOD,AND,SHL,SHR)
addOp (+,–,OR,XOR)
Low compareOp (<,=,>,<=,>=,<>,IN)

At the highest level, unary operators and arguments are evaluated. The evaluation order of the operators at the same priority level is not necessarily from left to right. The compiler can rearrange such expression to generate faster code. The only operators guaranteed to evaluate from left to right is OR and AND when they are used with Boolean arguments. Parentheses can always be used to assure correct evaluation order, as expressions entered within parentheses are always evaluated at the highest priority level.

Constant Expressions

Some operators and functions can be used on constant arguments and also return a constant value. The operators and functions allowing this are shown when they are described below.

constExpr:	expression

Binary Operators

Binary operators always take two arguments. The types of the arguments must be expression compatible to each other. The conditions for this compatibility differs with the operators and types, and are described at each type of operator.

Compare operators always generate code, and therefore cannot be used in constant expressions.

In the following description of operators, the value of the left argument is called leftArg, and the value of the right side argument is called rightArg.

Arithmetic Operators

The arithmetic operators are used on arguments of an integer or real type. Both types can be used together with one operator. If either operator is of real type, the operator is evaluated as a real operator. Also, if the operator is the real division operator (/), the result is always a real result, even if both arguments are of integer type.

The following compare operators are allowed on integer and real types. They always return a Boolean result.

	Evaluated to True if:
<	leftArg is less than rightArg
=	leftArg is equal to rightArg
>	leftArg is greater than rightArg
<=	leftArg is less than or equal to rightArg
>=	leftArg is greater that or equal to rightArg
<>	leftArg is not equal to rightArg

Examples:

```
1 < 2
3 = 3.0
```

Integer Operators

The following operators are allowed on integer types. The result of the operation is also described.

+	leftArg added with rightArg
–	rightArg subtracted from leftArg
OR	logical OR between leftArg and rightArg
XOR	logical XOR between leftArg and rightArg
*	leftArg multiplied by rightArg
DIV	integer part of leftArg divided by rightArg
MOD	remainder of leftArg DIV rightArg
AND	logical AND between leftArg and rightArg
SHL	left shift of leftArg by rightArg bits
SHR	right shift of leftArg by rightArg bits

The actual result type is depending on the combined type of the argument types. This is done so the result type can hold any value allowed in both the type of the left side argument and right side argument. If either argument is of LongInt type, the resulting type is always LongInt. For other combinations, the following table show the resulting type.

Arg types	*ShortInt*	*Byte*	*Integer*	*Word*
ShortInt	ShortInt	Integer	Integer	LongInt
Byte	Integer	Byte	Integer	Word
Integer	Integer	Integer	Integer	LongInt
Word	LongInt	Word	LongInt	Word

If either argument is a constant value, the resulting type is assumed to be the same as the non-constant argument type. If both arguments are constant values, the operator is immediately evaluated, and no code is generated.

There is one small problem with integer operators: overflow. The DL-Pascal compiler cannot detect integer overflow. Since the resulting type will only depend on the argument types, and not their values, the result may be different from the expected one due to non-detected overflow. E.g. if a variable of Byte type, having the value 255, is added by a constant value 2, the resulting value will be 1 (the low-byte value of 257, since that is all there is room for in a Byte type variable).

The easiest solution to integer overflow is to use type casting whenever the result of an expression can be larger than the range allowed by the argument types. Type casting is described on page 169. For the example above, the statement

```
Word(byteVar)+2
```

will evaluate to a result with type Word and the correct value 257.

Division by zero is always detected. The error generation for this can be disabled by the compiler option {$F–}.

Examples:

```
2 + 3       gives 5
2 - 3       gives -1
5 OR 9      gives 13
5 XOR 9     gives 12
2 * 3       gives 6
9 DIV 5     gives 1
9 MOD 5     gives 4
5 AND 9     gives 1
5 SHL 2     gives 20
5 SHR 2     gives 1
```

Integer and Real Operators

When either argument is of real or integer type, the following operators are allowed. The result of the operation is also described.

+	leftArg added with rightArg
–	rightArg subtracted from leftArg
*	leftArg multiplied by rightArg
/	leftArg divided by rightArg

When both arguments are constant values, the operator is evaluated immediately, and a constant value is returned.

The result type is always the real type. Overflow from any part of a real expression, and division by zero, is detected at the end of the expression. The error testing can be disabled by the use of compiler option {$F–}.

Examples:

```
2.0 + 3.0   gives 5.0
2.0 - 3.0   gives -1.0
2.0 * 3     gives 6.0
9 / 5       gives 1.8
```

Boolean and Enumeration Operators

The enumerated types can have a number of compare operators. The Boolean type can also be used with some other operators.

For comparison, the following compare operators are allowed. The result is always Boolean.

	Evaluated to True if:
<	leftArg is less than rightArg
=	leftArg is equal to rightArg
>	leftArg is greater than rightArg
<=	leftArg is less than or equal to rightArg
>=	leftArg is greater that or equal to rightArg
<>	leftArg is not equal to rightArg

The comparison is done according to the ordinal value of the enumerated identifiers.

Example:

```
False < True
```

In addition, the following binary operators are allowed on variables of Boolean type, returning a Boolean result.

	Evaluated to True if:
AND	leftArg and rightArg are True
OR	leftArg or rightArg is True
XOR	leftArg is not equal to rightArg

For binary operators AND and OR, the compiler generates *short-circuit* expressions. This means that if part of an expression is evaluated so the remainder of the expression will not change the result, the remainder of the expression is skipped. This is especially useful when the first part of an expression tests the value of a variable to be correct, and the second expression part uses that variable to index or point to another variable.

If it is needed for all parts of a boolean expression to be evaluated, the boolean expressions can be type casted to Byte type, and the complete expression type-casted to Boolean type.

All boolean operators generate code, and cannot be used in constant expressions.

Examples:

```
False AND True          gives False
False OR True           gives True
False XOR True          gives True
(ptr<>NIL) AND (ptr^.val>0)
```

The last example will use the short-circuit feature to skip the second expression part if the pointer variable ptr has the value NIL. In such a case, the first part will be evaluated to False, and as the second part cannot change that, it will be skipped. If it would not be skipped, and error testing was on, a run-time error would be generated.

String and Char Operators

On string arguments the string operators are used. The arguments can be of any string type, and also the Char type.

The result type from the compare operators is always Boolean. The following compare operators are allowed on string arguments.

	Evaluated to True if:
<	leftArg is less than rightArg
=	leftArg is equal to rightArg
>	leftArg is greater than rightArg
<=	leftArg is less than or equal to rightArg
>=	leftArg is greater that or equal to rightArg
<>	leftArg is not equal to rightArg

When comparing strings, the ordinal values of each character in the strings are compared. The actual ordering of characters can be seen in appendix D. This results in a upper case letter having a smaller ordinal value than a lower case letter.

If both strings are identical, the strings are equal. If two strings of different lengths are compared, and the strings are equal up to the length of the shorter one, the shorter string is the smaller one.

To make both lower and upper case letters be treated as the same letter, both strings should be converted to all upper case letters. If you also want national characters to be sorted in the correct sequence, the characters of each string should be converted into a value corresponding to the sort order for that country (as this varies between different countries).

Examples:

```
'A' < 'a'
'ABC' = 'ABC'
'ABCD' > 'ABC'
```

There is one other binary operator allowed on string types. It is the add operator, which is used to concatenate two strings.

+	rightArg added to the end of leftArg

The resulting type is the system string type (i.e. the resulting string can hold a maximum of 255 characters). Any characters past this limit are truncated.

The concatenation operator will always generate code, and therefore cannot be used in a constant expression.

Example:

`'ABC' + 'DE'` gives `'ABCDE'`

Set Operators

For set arguments, the set operators are used. The argument types must be of identical types, or one of the arguments can be an empty set ([ ]). The exception to this is the IN operator, where the right argument type must be a set type having the left argument type as the base type (SET OF leftArg type).

The compare operators allowed on set types are the following. The result type is always Boolean.

	Evaluated to True if:
=	both sets are identical
<>	the sets are different
<=	all elements of leftArg is also in rightArg
>=	all elements of rightArg is also in leftArg
IN	leftArg element is in rightArg set

Examples:

```
[0,1,2] <= [0,1,2,3]
[0,1,2] >= [1,2]
2 IN [0,1,2]
```

There are also three other binary operators allowed on set types. The result type is the same type as the argument types. The result of the operator is also described.

+	elements found in leftArg or rightArg (or both)
–	leftArg elements less the elements in rightArg
*	only elements found in both leftArg and rightArg

With all set operators, code is always generated. They cannot be used in constant expressions.

Examples:

`[0,1,2] + [2,3]` gives `[0,1,2,3]`
`[0,1,2] - [2,3]` gives `[0,1]`
`[0,1,2] * [2,3]` gives `[2]`

Pointer Operators

The pointer operators are used on arguments of pointer type. Both arguments must be of the same type, or either argument may be a system type (including the reserved word NIL).

Only the following compare operators are allowed on pointer types:

	Evaluated to True if:
=	both pointers point to the same memory address
<>	pointers point to different memory addresses

By using type casting to and from Word or LongInt types, the operators allowed on integer types can be used also with pointers. In this case the compiler assumes that the programmer knows what he (or she) is doing.

Unary Operators and Arguments

The unary operators and arguments allowed in a DL-Pascal program are:

factor: sign factor | NOT factor | (expression) |
number | string | NIL | constId | enumId | [set] |
& refAddr | @ refAddr | refOp

Arithmetic Operators

The unary arithmetic operator are unary plus and minus. They are allowed on integer and real types. The resulting type is the smallest one to hold the possible range of result values. Negation on the Byte and Word types will result in types Integer and LongInt returned, respectively.

The unary plus is included for symmetry with unary minus. It will not generate any code.

In addition, one's complement is allowed on integer types, by the reserved word NOT.

The arithmetic operators can be used in constant expressions, if the argument used with them is a constant value.

Examples:

```
+1
-2
NOT 13
```

Boolean Operator

On arguments of Boolean type, the NOT operator is allowed. This operator will return the inverted value of the argument following the operator (False to True and vice versa).

The NOT operator can be used in constant expression, if the corresponding argument is a constant.

Example:

`NOT False` gives `True`

Subexpression

A subexpression can be used as an argument, by surrounding the subexpression with parentheses. It can be used to regroup evaluation order in the expression, as the subexpression will always be evaluated before the operator connected to it.

Examples:

```
(1+3)
(4*(max-val))
```

Values

Arguments can also be constant values. Such values can be directly entered numbers and strings, but also a constant identifier previously declared in the CONST declaration.

The value can also be an enumerated identifier, in which case the type of the value is the enumerated type where the identifier was declared.

The last value is the reserved word NIL. It is used to signal that a pointer does not point to anything. The type of this value is the system pointer type.

Set Value

A set can be used as an argument. The elements of the set is entered within square brackets. If no elements are entered, the set is *empty*, and that set type is compatible to any other set. In all other cases, the type of the first element specifies the type of the set, and the type of all other elements must be compatible to the first element type.

set: [setExpr { , setExpr }]
setExpr: expression [.. expression]

As long as the elements are constant values, the resulting set is also a constant. This will allow sets to be used in constant expressions.

Examples:

```
[]
[1,2,4..7]
['Y','y','N','n']
```

Address Operators

There are two address operators: & and @. The only difference between them is how subroutine variables are decoded.

The & address operator always returns the address to the variable or subroutine given as the argument. The address can be thought of as a pointer to that variable or subroutine. The type of this address is the near system pointer, or the far system pointer if the variable is declared to be outside of the normal RAM area.

For variables not of subroutine type, and actual subroutines, the @ address operator works in the same way as the & operator. If the argument is a variable of a subroutine type, however, this operator returns the value in the subroutine variable (i.e. the address to the subroutine that would be called from this variable). In this case, the type is of far system pointer type.

This special handling of subroutine variables is needed when the current subroutine set in the variable has to be read. The problem is that a subroutine variable can be used to call the subroutine, without using parentheses after the variable. It is difficult, if not impossible, for the compiler to decide when a call to the subroutine is wanted, and when just the value of the variable is needed.

Since the address of a variable or subroutine is not known at compilation time, these operators cannot be used in a constant expression. In addition, some subroutines are decoded in a special way and cannot be used at all with an address operator. These functions are described on page 172, and procedures on page 183.

Examples:

```
&var1
IF @Proc=@procVar THEN ...
```

In the last example, the statement is executed if subroutine variable procVar currently is pointing to the Proc subroutine.

Variable Reference

In DL-Pascal, there are three different levels of variable referencing: *operator reference*, *address reference*, and *subroutine reference*.

refOp: refExpr { callDeref }
refAddr: refType { refDeref }
refProc: refExpr { refDeref }

The first part of the variable reference is used to specify a variable or subroutine identifier.

refExpr: varId | typeId (expression)
refType: varId | typeId (varId)

This part either specifies the variable directly, or by a *type conversion* or *type casting*. The type casting converts the actual type of a variable or expression into another type. In this way, the strict type checking of the Pascal language can be relaxed when needed.

There are two different types of type conversion: *variable* and *value* type conversion.

Value Type Conversion

Conversion of an expression is possible at operator and subroutine reference. Value type conversion is possible only if both the expression type and the destination type is of ordinal, pointer, or subroutine type, or a set type with a maximum number of elements of 16 or less. In all other cases, variable type conversion is used.

Examples (ptr and var are variables):

```
Word(ptr)
NormProc(var+3)
```

Variable Type Conversion

In all references, variable type conversion is possible. The size of both the source and the destination types must be identical. The source variable can also be an open array or an untyped subroutine parameter. In all other cases a compilation error is generated.

Example:

```
TYPE FullSet = SET OF Byte;
VAR  arr32 : ARRAY [0..31] OF Char;
...
FullSet(arr32)
```

In this example, both the source variable type and the destination type uses 32 bytes for storage, and therefore variable type conversion is allowed.

Variable Dereference

The second part of a variable reference is the dereferencing of the variable. This is used to select the wanted element of the variable.

callDeref: refDeref | callList

refDeref: arrayDeref | recordDeref | pointerDeref

The types that can be dereferenced are array, string, record, and pointer types. In addition, a function can be called at operator reference. The dereferencing can be nested (e.g. to get a record field, where the record itself is in an array, the record is first dereferenced and then the record field. All this can be done in one variable reference).

Array and String Dereference

When an array is dereferenced, one element in the array will be selected. A string dereference is used to extract one character in the string.

arrayDeref: [exprList]

exprList: expression { , expression }

For a string, the character is selected by a value specifying the offset in the string. The length byte is at index 0, and the first character at index 1. The value must be of integer type.

With an array, the element is selected by an array index. This index must be within the limits set when declaring the array type. The type of the index value must also be compatible with the declared index type.

A multi-dimensional array can be dereferenced either by surrounding all index values by square brackets, or by separating the indexes by commas, and surrounding the complete list by square brackets.

Examples:

```
VAR
   s : STRING[10];
   a : ARRAY[1..9,1..5] OF Char;
...
s[1]
a[3]
a[9][5]
a[9,5]
```

The first example gives the first character of the string (as type Char). The second one gives the third subarray of the variable (of type ARRAY [1..5] OF Char). The last two examples both return the last character of the array (of type Char).

Record Dereference

A record is dereferenced to select one field of the record variable.

recordDeref: . fieldId

The type set by a record dereference is the type of the record field. If this is a record itself, the fields of that record can be dereferenced by another record dereference.

Examples:

```
VAR r:RECORD
        f1,f2:Word;
        f3:RECORD f31:Byte END;
      END;
...
r.f1
r.f3.f31
```

The first example returns the field f1 of the record variable (type is Word). The last example gives the Byte field of field record f3.

Pointer Dereference

When pointers are dereferenced, the variable being pointed to is selected.

pointerDeref: ^

The result type of a pointer dereference is the type of the variable pointed to. A system pointer type can also be dereferenced; the result type is then the same as an untyped parameter, and must be type-casted to be usable. If the pointer being dereferenced has the value NIL, a run-time error is generated. This error test can be removed by compiler option {$T–}.

Examples:

```
TYPE PtrT=^Word;
VAR
  p1:PtrT;
  p2:^PtrT;
...
p1^
p2^
p2^^
```

The examples assume that variables p1 and p2 have been initialized with a value or address before they are used.

The first example will return the Word variable being pointed to. The second example returns the pointer variable that p2 is pointing to (in the declarations, p2 is declared to be a pointer to type PtrT, which itself is a pointer). The last example will get the actual Word variable being pointed at.

The pointer character (the caret) is used before declaring a pointer type (in the declarations), and after the variable when dereferencing a pointer. You can remember the position of the caret easily if you read the declaration as Pointer To type, and the dereference as variable Was Pointing.

Function Call

A function or function variable can be called in an operator reference.

callList: [callParams]
callParams: (callElement { , callElement })
callElement: expression | refAddr | refProc

Each parameter in the function declaration must be given a corresponding parameter value when calling the function. The called parameter value is different depending on the declared parameter type. For value parameters, an expression is entered, except for subroutine value parameters where a procedure reference must be entered. For variable, open array, and untyped parameters an address reference is entered.

Some functions declared in the System unit are decoded in a special way and have no address (and cannot be used with address operators). These functions are:

`Abs`
`Addr`
`Chr`
`Concat`
`Copy`
`Eof`
`Eoln`
`FilePos`
`FileSize`
`Hi`
`IOResult`
`Length`
`Lo`
`Max`
`Min`
`Odd`
`Ofs`
`Ord`
`Pos`
`Pred`
`Ptr`
`Random`
`Round`
`SeekEof`
`SeekEoln`
`Seg`
`SizeOf`
`Sqr`
`Succ`
`Swap`
`Trunc`

The functions Max, Min, and SizeOf will always return a constant value. Functions Abs, Chr, Hi, Length, Lo, Odd, Ord, Pred, Round, Sqr, Succ, Swap, and Trunc will return a constant value if the parameter is a constant value. All other functions will generate code, and they cannot be used in a constant expression.

The actual declarations and usage of the functions in the System and Crt units are described in chapter 31 and 32.

Chapter 24

Statements

In Pascal, the *statement* is used to assign values to variables, to call procedures, and for program execution control. In addition, a statement can be empty (i.e. nothing is to be executed in that statement). The empty statement allows the use of superfluous semicolons.

statement: [labelId :] [statElement]

statElement: compStat | ifStat | caseStat | repeatStat | whileStat | forStat | withStat | gotoStat | inlineStat | assignStat | procCallStat

A label can be set at a statement, by entering the label identifier followed by a colon before the statement. Execution can then be continued at that statement by the GOTO statement.

Compound Statement

Some control statements only allow a single statement to follow. In such cases, a sequence of statements can still be used by using a *compound statement.*

compStat: body

The syntax of the compound statement is identical to the program body (the main part of the program).

Example:

```
IF condition THEN BEGIN
  statement1;
  statement2;
END;
```

Both statement1 and statement2 will be executed if condition is True.

If Statement

The IF statement is used when a statement should only be executed when a certain condition exists.

ifStat: IF expression THEN statement [ELSE statement]

The conditional expression after the IF keyword must be a boolean expression. If this evaluates to True, the statement following THEN is executed. If evaluated to False, the statement is skipped.

The ELSE keyword can be used to execute a statement if the expression is evaluated to False. In this case, the ELSE statement is skipped if the expression is True.

An ELSE statement is always connected to the previous open IF statement (not terminated by an ELSE keyword or in any other way). This must be kept in mind when using nested IF statements. The construct

```
IF cond1 THEN
 IF cond2 THEN stat2
ELSE stat1;
```

will not be executed as the indentation would like one to think, as the ELSE part binds to the previous IF (the one using cond2).

In order to bind the ELSE part to the first IF, the second IF statement needs to be terminated. A semicolon cannot be used, as this would terminate both IF statements. Instead, this can be made by using a compound statement:

```
IF cond1 THEN BEGIN
  IF cond2 THEN stat2;
END ELSE stat1;
```

or by using an empty ELSE statement for the second IF:

```
IF cond1 THEN
 IF cond2 THEN stat2 ELSE
ELSE stat1;
```

In the first of these solutions, the second IF statement has a semicolon after it. In this case it is fully possible, since the THEN statement of the first IF is the compound statement BEGIN...END. Any number of statements can be included within the compound statement. The compound statement will still be counted as only one statement.

Case Statement

When different statements are to be executed depending on the value of one variable or expression, the CASE statement is used. If can be thought of a sequence of IF statements, testing the same variable or expression against different constant values.

caseStat: CASE expression OF
caseList [;]
END

caseList: caseEntry [; caseEntry] [caseElse]

caseElse: [;] ELSE statList

caseEntry: caseElement : statement

caseElement: caseRange { , caseRange }

caseRange: constExpr [.. constExpr]

The *control expression* following the CASE keyword must be an ordinal type, except for the LongInt type. The *control elements* of the CASE statement must be constant expressions of the same type as the control expression. Control elements are used to specify the statement to be executed. The control element matching the value of the control expression will select the statement to execute (the statement following the control element). If no control element is found, execution continues at the ELSE part of the CASE statement, or the CASE statement is skipped altogether if no ELSE part has been entered.

By using compound statements, more than one statement can be executed at a control element.

The control element can be a single constant, a list of constant values, or a range of constant values (or a combination of these). Code for the CASE statement is generated differently, depending on the type of control elements used. This will minimize the code size when values of the control elements lies far from each other.

In standard Pascal, it is an error if the value of the control expression is not found in the CASE statement. This restriction is not used in DL-Pascal.

Example:

```
CASE val OF
1: stat1;
2..9: stat2;
ELSE stat3;
END;
```

Either of stat1, stat2 or stat3 is executed depending on the value of val. If the value is 1, stat1 is executed. If the value is in the range 2 to 9 (inclusive), stat2 is executed. For all other values of val, stat3 is executed.

Repeat Statement

The REPEAT statement is used when a sequence of statements has to be repeated one or more times. The control expression of this repetition must be a boolean expression.

```
repeatStat:        REPEAT
                     statList
                   UNTIL expression
```

The statements enclosed within the REPEAT...UNTIL block are first executed one time. The boolean expression is then evaluated, and if the value of it is False, the statements are executed one more time. This is repeated until the expression is evaluated to True.

Example:

```
i:=1;
REPEAT
  Write(i);
  i:=i+step;
UNTIL i>=10;
```

While Statement

Another repetition statement is the WHILE statement. It works as the REPEAT statement, but the boolean control expression is evaluated at the beginning of the loop. This allows the statement in the WHILE loop to be skipped (i.e. repeated zero or more times).

```
whileStat:         WHILE expression DO statement
```

The control expression is evaluated, and if the value is True, the statement is executed. This is repeated until the expression evaluates to False.

If more than one statement is needed in the WHILE loop, a compound statement can be used.

Example:

```
i:=1;
WHILE i<maxVal DO BEGIN
  s[i]:=' ';
  i:=i+step;
END;
```

For Statement

When the number of repetitions is known at the start of the loop, and a single incrementing or decrementing variable is sufficient, the FOR statement can be used.

forStat:	FOR varId := startExpr direction endExpr DO statement
direction:	TO | DOWNTO

The *control variable* controlled by the FOR loop must be a simple variable (not part of a structured type), and declared at the current subroutine level or as a global variable. It's type must be an ordinal type (including a LongInt type).

A compound statement can be used if more than one statement is to be controlled by the FOR loop.

The FOR statement can be rewritten using a REPEAT construct:

```
varId:=startExpr;  endVal:=endExpr;
IF varId<=endVal THEN
 REPEAT
   statement;
   endCond:=varId=endVal;
   IF NOT endCond THEN INC(varId);
 UNTIL endCond;
```

The direction of the loop variable is set by the TO and DOWNTO keywords. The variable is incremented when the TO keyword is used, and decremented when the reserved word DOWNTO is used. When decrementing, the FOR statement can be rewritten as:

```
varId:=startExpr;  endVal:=endExpr;
IF varId>=endVal THEN
 REPEAT
   statement;
   endCond:=varId=endVal;
   IF NOT endCond THEN DEC(varId);
 UNTIL endCond;
```

If the FOR loop was not executed at all (if the start expression value was larger than the end expression value, or smaller for decrementing), the control variable will be assigned the start expression value when execution gets past the FOR statement. When the loop terminates after being executed at least once, the control variable will have the value of the end expression.

Examples:

```
FOR i:=1 TO 10 DO arr[i]:=i;
FOR val:=254 DOWNTO Length(s) DO BEGIN
  IF (s[val]=0) AND (s[val+1]=0) THEN s[val]:=255;
END;
```

With Statement

This statement is used to allow the fields of a record variable to be easily accessed. The fields of the record dereferenced by the WITH statement can then be used as normal variables.

withStat: WITH refAddr { , refAddr } DO statement

The address reference must return a record variable. The easier access to the record's fields is then valid in the following statement. Normally, a compound statement is used as the statement to allow more than one statement to see the record fields.

More than one record variable can be entered in one WITH statement. This has the same function as a sequence of WITH statements. The construct

```
WITH rec1 DO WITH rec2 DO stat1
```

can be more easily entered as

```
WITH rec1,rec2 DO stat1
```

The WITH statement is often used when record variables are part of another variable, or when the record is used together with pointers. As the reference to the record is only calculated once, the execution will be faster than if the reference would have to be recalculated for each access to the field of the record.

Example: The code sequence

```
rec[i].val:=5;
rec[i].count:=1;
rec[i].name:='Name';
```

will execute faster and take less space if it is rewritten as:

```
WITH rec[i] DO BEGIN
  val:=5;
  count:=1;
  name:='Name';
END;
```

If there is a local or global variable with the same name as a field identifier accessible within the WITH statement, the field identifier will take precedence. The local variable then cannot be accessed within the WITH statement. A global variable can be accessed in the same way as another unit's variable: by prefixing the global variable name by the program identifier (see page 155).

Example:

```
PROGRAM Prog1;
TYPE R1=RECORD
          v1:Integer;
          ...
        END;
VAR v1:Word;
    rec1:R1;
PROCEDURE P1;
  VAR v1:Char;
  BEGIN
    WITH rec1 DO BEGIN
      v1:=1;           {record field}
      Prog1.v1:=2;     {global variable}
    END;
    v1:='A';           {local variable}
  END;
...
```

Within the WITH statement, v1 will select the record field. When prefixed by the program name, v1 will specify the global variable.

Goto Statement

The GOTO statement is used in conjunction with labels to continue execution at another statement in the current subroutine or main program.

gotoStat: GOTO labelId

This statement is very seldom used in a Pascal program. The only real reason to use it is to jump out of nested control statements due to an error or termination condition.

The jump can only be made within the current subroutine, or within the main program part. It is not possible to continue execution in another subroutine with the GOTO statement, as is possible in standard Pascal.

Jumps can be made out from FOR and WITH statements without problem. Jumps into such statements should not be made, as the result is then undefined.

Examples:

```
LABEL OutJump;
...
  FOR i:=1 TO 10000 DO BEGIN
    ...
    IF a[i]=255 THEN GOTO OutJump;
    ...
  END;
OutJump:
  ...
```

Inline Statement

It is possible to add machine code to a Pascal program by using the INLINE statement. It can also be used to set data in a program to be specially treated by the program. The normal use, however, is to make machine code subroutines for code parts that need to run as fast as possible.

inlineStat: INLINE (inlineData)
inlineData: inlineExpr { , inlineExpr }
inlineExpr: [sizeSpec] inlineValue
sizeSpec: < | = | >
inlineValue: constExpr | id { sign constExpr }

The inline code is entered as values within the INLINE statement. Each value is normally one byte in size if the value is less than 256, and otherwise two bytes. Each value is separated from each other by a comma. The machine code for various assembler instructions are shown in appendix G.

In addition to constant expressions, identifiers can be entered. The value inserted to the INLINE statement is the offset from the IX register (for local variables and parameters), or the actual address (for global variables and subroutines). The identifier can also have a constant expression following it, to adjust the offset to the variable. The size of an entry of this type is normally two bytes.

Both expressions and variables can have a prefixing size specifier, to change the normal size. By a less-than sign (<), the size is always one byte, even if the value is larger than 255. A larger-than sign (>) is used to assure the value uses two bytes. With an equal sign (=), the size is also one byte, but the value set is the negated value of the expression entered. This is handy when using local variables having a negative index from register IX.

Global and absolute variables, and subroutines, must have a size of two bytes. A compilation error is generated if another size specifier has been entered.

Examples:

```
INLINE($D1, 8, >240);
INLINE(<lvar, =lvar, >lvar-2);
```

If lvar is a local variable at offset IX–4, the following code will be generated by the INLINE statements:

```
$D1, $08, $00F0
$FC, $04, $FFFA
```

Assignment Statement

To assign data to variables, the assignment statement is used. There are two different decodings of an assignment, depending on the type to be assigned.

```
assignStat:     id := refProc |
                refAddr := expression
```

If the assignment is to a subroutine variable, no expression is allowed, only a procedure reference. For all other types, an expression is decoded. The assignment is allowed to all types, even record and array types. Assignment compatibility must, however, be satisfied.

Examples:

```
VAR graphCall:FUNCTION (arg:Real):Real;
    s:STRING;
...
  graphCall:=Sin;
  s:='Added '+'string';
...
```

Procedure Call

Procedures can be called from a statement. Both normal procedures and procedure variables can be used to call the procedure.

procCallStat: id callList

The replacement of parameters for the procedure is the same as replacement to functions. This is described on page 172.

There are a number of procedures in the System unit that are decoded in a special way. These procedures cannot be used with the address operators:

`Append`
`Assign`
`BlockRead`
`BlockWrite`
`Close`
`Dec`
`Delete`
`Dispose`
`Erase`
`Exit`
`FillChar`
`Halt`
`Inc`
`Insert`
`Move`
`New`
`Page`
`Read`
`ReadLn`
`Rename`
`Reset`
`Rewrite`
`RunError`
`SetTextBuf`
`Seek`
`Str`
`Val`
`Write`
`WriteLn`

The actual declarations and usage of the procedures in the System and Crt units are described in chapters 31 and 32.

Chapter 25

Interface and Implementation Units

In DL-Pascal, it is possible to build library files, where often-used subroutines can be placed and used by different programs. Such files are called *unit files*. This ability is an enhancement to standard Pascal.

The dynamic linking used in DL-Pascal allows a number of programs to share the same unit file. In this way, the additional storage needed for a new program can be very little, if the new program only uses units already in use by other programs.

A unit is separately compiled from the program using the unit. As long as the declarations of the unit do not change, there is no need to recompile the unit. This can also be used to split a large program into different units, to reduce the compile time when only small changes will be made to the program. If a program is split into four units, each unit's compilation time takes only one-fourth of the time the original large program took. For small changes in the program (i.e. in only one unit), the time turn-over to run the changed program will be faster by four times.

Different information about a unit is needed at compilation and at run time. To reduce memory consumption, a unit is split into two parts: the *interface part* and the *implementation part*. The interface part is needed when other units and programs using that unit is compiled. The implementation part contains the actual code, and that part is needed when the program using the unit is being run.

Interface Unit

In the interface unit, all identifiers to be visible to other programs are declared. The interface file must be compiled before the unit can be used by other programs. The compilation generates a code file using the name of the interface file, but with extension **UNI** (for **UNI**t file) and file type **U**.

interfaceDecl: INTERFACE newId ;
[usesDecl]
declInt
END .

The identifier entered after the reserved word INTERFACE is the name of the unit. That name and the file name of the unit should be the same (the file name being the first eight characters of the unit name, and in upper case), otherwise the computer may later have problems finding the unit. The file name **must** be in upper case, as the Pascal compiler automatically converts all identifiers to upper case.

All units used by the interface declarations are specified in the USES declaration, if needed. This is used as in a normal program.

The declarations of an interface file are a bit different from a normal program file:

declInt: { oneDeclInt }
oneDeclInt: constDecl | typeDecl | varDecl | procFuncHead

Constant, type, and variable declarations are identical to a normal program. Declaration of subroutines are different from a program declaration. In the interface unit, only the subroutine heading is declared. The body of the subroutine is later declared in the implementation unit. This can be thought of as forward-declaring the procedures and functions in the interface (although the FORWARD keyword must not be used).

procFuncHead: PROCEDURE newId procHead ; |
FUNCTION newId funcHead ;

The specification of parameters and return type is done as in a normal procedure or function.

An interface unit has to be recompiled only when it has been changed. At that time, all other units and programs using that unit must also be recompiled. A version number (using a checksum of the declared identifier table) is used to avoid calling a unit which interface part has been changed. If such an attempt is found, a compilation or run-time error is generated.

Implementation Unit

The actual implementation (declarations and main part) of the subroutines declared in the interface is entered in the implementation unit.

```
implementDecl:  UNIT newId ;
                [ usesDecl ]
                declarations
                body .
```

Except for the starting reserved word UNIT, the structure of the implementation unit is identical to a normal program. When the unit is compiled, it will generate a link file with the same name as the source file, but with extension **DLE** (Dynamic Link Executable) and file type **E**.

The main part of a unit is used for initialization of that unit. It will be called before the other units and the program uses the unit (except for circular import; see below).

All identifiers declared in the interface unit are automatically imported into the implementation unit. In addition, all units imported to the interface file are automatically visible in the implementation file. This also holds for any units indirectly imported by the interface unit (i.e. units imported by the interface might themselves import other units. Those units will automatically be visible in the implementation unit).

The actual declaration of subroutines is done in the same way as a declaration of a previously forward-declared subroutine, in that only the name of the subroutine is needed (not the parameter declaration), as described on page 156.

Using Units

To use an identifier declared in another unit, that unit must first be imported into the current unit or program. This import is done by the USES declaration, described on page 128.

Chapter 4 in part one of the manual describes an example using units in different programs.

Circular Import

When writing units importing other units, sometimes two (or more) units will import each other. This is called *circular import*. This is allowed in DL-Pascal, but can be a problem if the initialization code (the main part) of one unit needs a variable in the other unit, and that variable must first have been initialized by the main part of that unit. It is difficult to detect in advance which unit is initialized first.

System and Crt Units

There are two predeclared units in DL-Pascal. They are the System and Crt units. The System unit is automatically imported into every program and unit being compiled. The Crt unit has to be manually imported when an identifier declared in it is needed.

All identifiers declared in the System and Crt units are described in chapters 31 and 32.

Chapter 26

Dynamic Memory Allocation

In addition to reserving memory space directly in the program for global and local variables, space can be allocated dynamically during execution of a program. This is useful if the size of the variable is not known at compilation time. Also, if a number of variables is needed, but the exact number not known at compilation, the *dynamic memory allocation* can be used.

The *heap* is used to store variables allocated in the dynamic memory. In addition, the heap is used to store both global and local variables, and also parameters when calling subroutines.

The dynamic memory is allocated when the *dynamic variable* is needed, and released to the system when the variable is no longer needed. This makes it possible to use more variables than possible with global variables, if only a few variables are used at one time.

Variables allocated in this way are accessed by using pointers. Normally, the pointer variable is declared as being a pointer to the type of variable that will be allocated in dynamic memory. Alternatively, the system pointer type can be used. In this case, type casting is needed to use the variable. This should only be used in special cases, as the type checking is relaxed when type casting is used.

There are two different ways to release the allocated memory. The first way is by using Mark and Release, and the second way by using Dispose or FreeMem. These two ways should not be combined within one program (including all units used by the program).

The allocated memory must be released when the allocated variable is no longer used. If not released, it will use that memory space forever (well, until the end of program execution). This will result in very little dynamic memory being available for allocation of other variables after some time.

When there is no memory left for any more dynamic variables, a run-time error is normally generated. This can be changed by the program, however, by using the heapError function in the System unit.

New and GetMem

Dynamic memory is allocated by using the New or GetMem procedure. The difference between these procedures is that for the New procedure, the allocated size is automatically derived from the base type of the pointer type. For GetMem, the size must be explicitly given. In fact, procedure New can be rewritten using GetMem as:

```
GetMem(ptr,SizeOf(ptr^))
```

Normally, GetMem is only used when the actual size to allocate is not known when the program is compiled, or when the allocated memory is to be used as a special variable (e.g. as a file buffer).

Example:

```
TYPE
  StrType =STRING;
VAR
  p1 :^StrType;
  p2 :Pointer;
BEGIN
  ...
  New(p1);
  GetMem(p2,256);
  p1^:='String data';
  Word(p2^):=1;
  ...
END.
```

In both cases, 256 bytes of memory will be allocated (in the first case because the system string variable needs 256 bytes). When assigning a value to the second variable, type casting is needed, as the system pointer type was used as the pointer type. A compilation error would occur if the type casting is omitted when using a dereferenced system pointer type.

Mark and Release

The Mark and Release procedures can be used to release allocated memory both simple and fast. These procedures are used to mark a group of allocated variables to be simultaneously released.

The Mark procedure is used to read the current end of allocated memory, and must be called before allocating any variables in this group. When the variables are no longer needed, the Release procedure is used to release the allocated variable group.

Both the Mark and Release procedures take a parameter, which is used to store the current end of allocated memory. In this way, any number of variable groups can be used, if the current memory end variable are stored in different variables.

Example:

```
TYPE
  StrType =STRING;
VAR
  saveP :Pointer;
  p1,p2 :^StrType;
BEGIN
  ...
  Mark(saveP);
  ...
  New(p1);  New(p2);
  p1^:='First string';
  p2^:='Next string';
  ...
  Release(saveP);
  ...
END.
```

When Release is called, the allocated memory for both variables p1 and p2 is released.

Dispose and FreeMem

The allocated variables can also be released one by one. This is the use for the Dispose and FreeMem procedures. The FreeMem procedure takes a second parameter which is the same size as used when allocating memory for that variable. Also here, the Dispose procedure can be rewritten using:

```
FreeMem(ptr,SizeOf(ptr^))
```

An allocated variable must be released only once. If released again before being allocated, the internal structure of the dynamic memory can be destroyed.

Dynamic variables can be released in any order. To be able to reuse the released memory, a *free list* is built in the dynamic memory. These free areas in dynamic memory is reused for new allocated variables, whenever possible. When releasing memory, consecutive free areas are merged together.

The Release procedure should not be used in conjunction with Dispose (or FreeMem), as a call to this procedure will remove the free list. Any free areas in the dynamic memory will then be lost.

Example:

```
TYPE
  StrType =STRING;
VAR
  p1 :^StrType;
  p2 :Pointer;
BEGIN
  ...
  New(p1);
  GetMem(p2,256);
  ...
  Dispose(p1);
  FreeMem(p2,256);
  ...
END.
```

The memory allocated to pointer p1 is released by the call to Dispose, and the memory to pointer p2 is release by the FreeMem procedure.

MaxAvail and MemAvail

The free memory in the dynamic memory can be checked by these functions. The difference between them is that MaxAvail will return the largest block that can be allocated, while MemAvail returns the total free size in the dynamic memory.

Example:

```
VAR
  p1 :Pointer;
BEGIN
  ...
  IF MaxAvail<2000 THEN
   WriteLn('Not enough memory')
  ELSE BEGIN
    GetMem(p1,2000);
    ...
  END;
END.
```

The error message will be written if there is not enough memory to allocate space for the pointer.

HeapError

Normally when there is not enough memory in the dynamic memory for a new variable, a run-time error occurs. This handling can be changed by using the heapError procedure variable. This variable is declared as:

```
heapError :FUNCTION (size:Word) :Byte
```

The parameter size is the memory size to allocate. The function assigned to this variable can then try to make room for this memory space, e.g. by removing unneeded variables from the dynamic memory. It returns a Byte value, specifying the action to be taken. If returning value 0, a run-time error is generated (this is returned by the default heapError function). The value 1 is used to return to the program calling the New or GetMem procedure, while setting the pointer variable to NIL. Any other value will retry the allocation, supposedly because enough memory had been freed. If the memory still cannot be allocated, this function variable is called again.

By returning a NIL value to the pointer, instead of a valid address, the program calling New (or GetMem) can check if the memory could be allocated. If allocation was not possible, different action can then be taken from the one to make if allocation was successful.

Example:

To install a function to return the NIL value to a pointer when allocation cannot be made, the following program section could be used:

```
FUNCTION MyHeapError (size :Word) :Byte;
  BEGIN
    MyHeapError:=1;
  END;
...
VAR
  p1 :Pointer;
BEGIN
  heapError:=MyHeapError;
  ...
  GetMem(p1,1000);
  IF p1=NIL THEN
   WriteLn('No Memory')
  ELSE BEGIN
    ...
  END;
END.
```

Chapter 27

Files

Files are used to hold data for a program, when the data will be needed another time the program is run. Also, files on a floppy disk can be used when large amount of data is needed. One floppy disk can hold up to 320 KByte of data.

There are three different file types usable by a Pascal program: *Typed file*, *Untyped file*, and *Text file*. The file type to use depends on the needed usage of the file.

Before a file variable can be used, a file specifier must be assigned to that variable. This is done by the Assign procedure. The file specifier assigned specifies the device, file name, extension, and possibly also the file type. If no file type is entered, the default file type is S for a text file, and T for the other file variables.

To be able to read or write to the file, the file must first be opened. The procedures Reset, Rewrite, and Append are used for this purpose. The action taken by these procedures depends on the file variable type.

When the work on the file has been completed, the file should be closed by a call to the Close procedure. This will update the directory information on the disk or in memory. All files opened when the program terminates are automatically closed.

There are two other actions that can be taken on an assigned file. The first one is renaming the file to another file name. This is done by the Rename procedure. The second one is the Erase procedure, which will delete the file and it's contents from the disk or memory.

Up to 20 files can be open at one time. There are two restrictions regarding the usage of devices in different files. The first one is that only one file can use the CAS: device, and then no other file can use the COM: device. The second restriction is that when more than one file is using the COM: device, the first RS-232C file specifier will be used for all files using the RS-232C device.

IOResult Error Handling

If a file operation error occurs, a run-time error is normally generated. This can be disabled by the {$I–} compiler option. When this option is used in a program, any file errors will instead set an internal variable. This variable is then read by the IOResult function.

Any file operation requested after a file error will be completely skipped, until a call to IOResult is made. The error code can only be read one time by IOResult after a file error; after the first call, the no-error code (value zero) will be returned.

The run-time error generation of file errors can be re-enabled in any place in the program by using the {$I+} compiler option.

Typed Files

A typed file works like an array. The typed file is declared as a file of a certain type.

```
f:FILE OF type;
```

Each element in the typed file is accessed separately. A specific element can be accessed by using the Seek procedure. The elements in the file are numbered from 0 upwards. The current file position (the next element to read or write) can be determined by the FilePos function, and the current number of elements in the file by the FileSize function. If the file position is at the end of the file, the Eof function will return True. To place the file position at the end of the file (after the last element), you write:

```
Seek(f,FileSize(f))
```

To create the typed file, the file is opened by the Rewrite procedure. To reuse a previously created file, the Reset procedure is used instead.

The elements of the file is read by the Read procedure, and written by the Write procedure. More than one element can be read or written at the same time, simply by entering more than one parameter to the procedure. Only variables of the same type as the base type of the declared file variable can be used as the parameters to Read and Write.

Only the RAM and floppy disk can be used with a typed file.

Example:

The following program will use a typed file to store a name at the end of the file **NAME.DAT**.

```
TYPE
  DataType=RECORD
             name :STRING[30];
             phone:STRING[15];
           END;
VAR
  f :FILE OF DataType;
  d :DataType;
```

(continued)

(continued)

```
BEGIN {$I-}
  Assign(f,'NAME.DAT');
  Reset(f);
  IF IOResult>0 THEN Rewrite(f);
  ...
  d.name:='Name to set';
  d.phone:='123-456-7890';
  Seek(f,FileSize(f));
  Write(f,d);
  ...
  Close(f);
END.
```

Untyped Files

Files can also be accessed at a low level. For this purpose, untyped files are used. The declaration of an untyped file is done by the FILE keyword.

```
f:FILE;
```

When an untyped file is opened (by Reset or Rewrite), an optional *record size* parameter can be entered. This parameter specifies the size of a block to be used in that file. If the exact representation of a file is not known, this record size is usually set to 1 to be able to read and write every byte in the file.

Data in the file is read by the BlockRead procedure. The BlockWrite procedure is used to write data to the file. As with the typed file, the procedure Seek is used to set the file position, and the functions Eof, FilePos, and FileSize are used to check the file position and file size. The file position and size are expressed in record size blocks, so the file position in bytes is:

```
file position  *  record size
```

For both BlockRead and BlockWrite procedures, an optional variable parameter can be entered to return the actual number of complete blocks read or written. This variable can then be used to check that all wanted blocks have been read or written, and to do some special action if not. Partially read or written blocks (e.g. the last block in the file having a size of less than the record size) are not counted in this parameter.

Only the RAM and floppy disk devices can be used with an untyped file.

Example:

The following program will copy the contents of file **FILE1** to **FILE1.CPY**. The record size is set to 1, as every byte must be copied. Also, the destination file type is set to sequential type.

```
PROGRAM FileCopy;
VAR
  f1,f2 :FILE;
  rd    :Word;
  buf   :ARRAY [0..255] OF Char;
BEGIN
  Assign(f1,'FILE1');
  Assign(f2,'FILE1.CPY;S');
  Reset(f1,1);  Rewrite(f2,1);
  WHILE NOT Eof(f1) DO BEGIN
    BlockRead(f1,buf,256,rd);
    BlockWrite(f2,buf,rd);
  END;
  Close(f1);  Close(f2);
END.
```

Text Files

The last file type is used to write text. A text file is declared to be of type Text, declared in the System unit. Strings, and integer and real numbers can be written to and read from a text file. In addition, boolean values can be written. Any device can be used in connection with text files.

A text file can only be opened in one direction at a time, either for input (by using the Reset procedure), or for output (by using either of Rewrite or Append procedures). The Append procedure is used to append data at the end of a previously written text file.

To write data to a text file, the Write or WriteLn procedure is used. The difference between these procedures is that in addition to the data written, the WriteLn procedure also writes a new-line character (to start the next write on a new line). The data is entered as a number of expressions to the procedure, separated by commas. Different data types can be mixed in one procedure. A hexadecimal number can also be written, if a LongInt number or variable is prefixed by a percent sign.

Data is read from a text file by using the Read or ReadLn procedure. After the specified data has been read, only the ReadLn procedure skips all non-read data until the next new-line character is found (the next line is found). The next read statement the starts reading at this next line.

The parameters entered to the Read/ReadLn procedures must be variables. Also here it is possible to read hexadecimal numbers by prefixing a LongInt variable by a percent sign.

When numeric data is read from the text file, all characters up to the next space or control character is read, and thereafter the number is converted. If some non-allowed characters is found, a file error is set.

When reading string data, all characters up to the new-line character are read, regardless of how large the string variable is declared to be. If not all characters can fit into the string, the excessive characters are truncated. To be able to read more data after a string has been read, the ReadLn procedure should always be used when reading a string variable from a text file.

The procedure Page is used to write a new-page character to the text file. Depending on the output device, this will clear the window on the display, or feed the next paper into the printer.

To check for the end of a text line, the functions Eoln and SeekEoln can be used. These functions return True if the new-line character is the next character to read. Similarly, the functions Eof and SeekEof are used to check for the text file end, in which case they return True. The difference between these functions is that SeekEoln and SeekEof first skip past any spaces and control characters (other than the character to test for), before checking the end-of-line or end-of-file status.

When reading from the keyboard, the end of file status can be set by pressing the ON BRK key (or S ON BRK in debugger mode).

The buffer for a text file variable is normally 64 bytes. This buffer size can be changed by the SetTextBuf procedure. The buffer is normally set to a larger size to speed up text file writing. When reading from character devices such as the RS-232C, the buffering of characters can be removed by setting the buffer size to 1 instead.

There are two predefined text files in a Pascal program. These are the input and output files. If no text file is specified with the text file procedures and functions, the input or output variable is automatically used (depending on whether to read from or write to the text file). Normally, the input variable is connected with the keyboard, and the output variable with the display. This can be changed when calling the program, by using *file redirection*. Redirection is described on page 61. Other file variables can use these files (including the redirection possibility) by assigning an empty name to the new file:

```
Assign(t,'');
```

When the Crt unit is used by a program, the output variable is automatically connected to a special text file for faster writing to the display. The redirection of the output variable is in this case deselected.

Example:

The following program encrypts a string read from the keyboard, by writing the next character code to the read one (except for spaces and control characters). This is repeated until the ON BRK key is pressed (setting the Eof status).

```
PROGRAM TextDemo;
VAR
  ch :Char;
BEGIN
  WHILE NOT Eof DO BEGIN
    WHILE NOT Eoln DO BEGIN
      Read(ch);
      IF ch>' ' THEN Write(Succ(ch))
      ELSE Write(ch);
    END;
    ReadLn;
    WriteLn;
  END;
END.
```

By using SeekEoln and SeekEof instead of Eoln and Eof, spaces and control characters will be skipped. Thus, the entered text will be written in one long string, without spaces.

User Text Type

To be able to use the powerful Read and Write procedures in more special ways, it is possible to define *user text types*. All text file operations can still be used, but the actual input or output of data will be made through a special function field in the text variable.

The text type can be thought of having the following declaration:

```
BufferType=ARRAY [0..63] OF Char;
TextFuncType=FUNCTION(VAR t:Text) :Byte;
TextType=RECORD
           handle  :Byte;
           mode    :Byte;
           opCode  :Byte;
           device  :Byte;
           fileType:Byte;
           fileName:ARRAY [0..10] OF Char;
           bufSize :Word;
           bufRead :Word;
           bufWrite:Word;
           bufAddr :^BufferType;
           textFunc:TextFuncType;
           res     :ARRAY [0..3] OF Byte;
           buffer  :BufferType;
         END;
```

To use a user text type, declare the text type as above (in a TYPE declaration), and the variable as a normal Text variable (VAR t:Text). The fields of the text file variable can then be accessed using type casting, e.g.

```
TextType(t).device
```

Assign to Text File

Before using a text file variable, some fields must be assigned specific values. This is the same function as made by the Assign and AssignCrt procedures. The fields that should be initialized are:

mode	0
bufSize	64
bufRead	0
bufWrite	0
bufAddr	&buffer
textFunc	MyTextFunc

In addition, the fields handle, device, fileType, and fileName can be assigned and used freely. These fields are not used by the system in a user text type.

The textFunc field is the function variable called to actually read and write data through the text file. The operation code for this function is given in the opCode field:

0	Open text file.
1	Close text file.
2	Read from text file.
3	Flush read at end of Read.
4	Write to text file.
5	Flush write at end of Write.

The value returned from the function is the error code, if an error occurred while reading or writing data. If no error was detected, the value zero must be returned.

The memory area at $1530 to $162F is used by the conversion routines in the Read and Write procedures. If calling these procedures within the user function variable, the contents of this area must be preserved.

Open Text File

A call to Reset, Rewrite or Append will call the function variable with the opCode field set to 0. In addition, the mode field will be set to 1, 2, or 3 respectively.

No special field needs to be updated by the open function code.

Close Text File

This operation code is used when the text file is closed. When the text file has been opened for output, the Write Flush command is sent directly before this operation code.

Also here, no special field update is needed.

Read from Text File

The data to be used by the Read procedure is read by this operation code. The read data must be placed in the buffer, pointed to by the bufAddr field.

The fields to update after reading are:

bufRead	offset to first character in buffer (normally 0)
bufWrite	number of characters in buffer

If the field bufWrite is returned having the value zero, the Eof status will be set for this text file. The next read will then call this function again.

Flush Read Text File

This operation code is called at the end of the Read procedure. It is normally not used by a text file, but can optionally flush the data currently read into the buffer. In this case, the following fields should be updated:

bufRead	0
bufWrite	0

Write to Text File

When the text file buffer has been filled, this operation code is called. It's purpose is to write the contents of the buffer to the output device.

The field bufWrite holds the number of bytes in the buffer. The first byte to write is set at the first position in the buffer.

After the data has been written, the following field must be updated:

bufWrite	0

Flush Write Text File

This function code is called at the end of a Write procedure, or before the text file is closed. Normally, this will only do the same as the previous command; namely to write the buffer contents to the output device. In addition, this command can execute some other function, possibly time-consuming, to actually make the characters appear on the output device.

When called immediately before closing the text file, the mode field will be set to zero.

Example:

An output text file type, that will remove all characters except spaces and digits, could be written as the following program (normally, the text file declaration and function would be entered in a separate unit, to be easily used by other programs).

```
PROGRAM UserTextDemo;
TYPE
  BufferType=ARRAY [0..63] OF Char;
  TextFuncType=FUNCTION(VAR t:Text) :Byte;
  TextType=RECORD
             handle   :Byte;
             mode     :Byte;
             opCode   :Byte;
             nameData:ARRAY [0..12] OF Char;
             bufSize :Word;
             bufRead :Word;
             bufWrite:Word;
             bufAddr :^BufferType;
             textFunc:TextFuncType;
             res      :ARRAY [0..3] OF Char;
             buffer   :BufferType;
           END;
FUNCTION OwnTextFunc (VAR t:Text) :Byte;
  TYPE
    SaveArea=ARRAY [0..255] OF Char;
  VAR
    i   :Integer;
    ch  :Char;
    tmp:SaveArea;
    sav:SaveArea ABSOLUTE $1530;
  BEGIN
    OwnTextFunc:=0;
    WITH TextType(t) DO
     IF opCode>=4 THEN BEGIN
       tmp:=sav;
       FOR i:=0 TO bufWrite-1 DO BEGIN
         ch:=bufAddr^[i];
         IF ch IN [' ','0'..'9'] THEN Write(ch);
       END;
       bufWrite:=0;
       sav:=tmp;
     END;
  END;
```

(continued)

(continued)

```
VAR
  t:Text;
BEGIN
  WITH TextType(t) DO BEGIN
    mode:=0;
    bufSize:=64;
    bufRead:=0;
    bufWrite:=0;
    bufAddr:=&buffer;
    textFunc:=OwnTextFunc;
  END;
  Rewrite(t);
  Write(t,'123 but 0 text.');
  WriteLn(t,32 DIV 3:3,'not this');
  Close(t);
END.
```

Chapter 28

Program Termination

A Pascal program is normally *terminated* when the last **END** of the program is reached. It is also possible to terminate the program earlier, e.g. because some parameters are missing, or an error has occurred. There are two procedures for such termination: Halt and RunError.

When the program terminates, either because of a call to these procedures, the reaching of the last **END**, or a run-time error, the execution stack will be cleared. This is done in case the reason for termination was stack overflow. The dynamic memory, however, is not changed.

Halt

This procedure will terminate execution, and sets the **exitCode** variable to the parameter value entered. If no parameter is given, the variable is set to zero.

When the program is terminated by Halt, no error message will be written. The ERR variable in **CAL** mode will be updated, though.

RunError

Program termination by RunError is similar to Halt, but both the exitCode and errorAddr variables will be set. The exitCode variable will be set to the parameter value (or zero if no parameter is given). Variable errorAddr will be set to the code address in the program, where the error occurred.

The address set in errorAddr will be used when the error message is written. The source file line number corresponding to the program code address will be looked up in the debug information. If debug information is found, both the program name and the line number will be added to the error message.

ExitProc Variable

The user can install his own *termination procedure* in a program by using the exitProc variable. This can be used e.g. to close files used by the program in a controlled way, possibly by updating some records in the file. The exitProc variable is declared as a parameterless procedure:

```
exitProc :PROCEDURE;
```

When the program is started, the system installs it's own termination procedure to the exitProc variable. This default procedure will write an error message if the value of the errorAddr variable is not NIL. The error message will be written according to the exitCode error code. Note that it is possible to skip the error message even when RunError is used for termination, by setting the errorAddr variable to NIL in your own termination procedure.

When adding a termination procedure, the previous exitProc variable should be saved in a procedure variable, and be reassigned to the exitProc variable at the end of your procedure. This will assure that all termination procedures will be called, regardless of the number of procedures installed. It also allows units to install their own termination procedures without problems.

If a run-time error occurs in one termination procedure, program execution terminates immediately. Therefore, all run-time error checking should be off within the termination procedure.

Example:

The following program will demonstrate the use of termination procedures, by writing a message when the Halt statement is reached.

```
PROGRAM HaltDemo;
VAR
  nextExit :PROCEDURE;
PROCEDURE MyExit;
  BEGIN
    WriteLn('MyExit, error:',exitCode:3);
    exitProc:=nextExit;
  END;

BEGIN
  nextExit:=exitProc;  exitProc:=MyExit;
  WriteLn('Program started.');
  Halt(2);
  WriteLn('This is not written.');
END.
```

Chapter 29

Windows

All reading from and writing to the display uses *windows* to restrict the output to certain areas of the display. The initial window is set to the full screen display. To change the window, the Crt unit must be imported to the program.

In addition to restrict output to a window, output can be restricted to just one line. This is done by setting value 2 to the dispStatus variable (also in the Crt unit). In this case, reading and writing will only use the current display line, without moving or scrolling to the next line.

Coordinates regarding windows and cursor position are numbered from 1 upwards, where (1,1) is at the upper left corner of the window.

Setting a Window

A new window is set by the Window procedure. The window is specified according to the screen coordinates, not to any previously set window.

The current window coordinates can be read by the WindMinX, WindMinY, WindMaxX, and WindMaxY functions. These functions can be used to save the previous window coordinates while using another window. After that window has been used, the previous window coordinates can be restored.

Example:

The following program part will save the current window position and set a new window to the upper two lines and the rightmost eight characters (position 25 to 32). Later, the saved window will be restored.

```
USES Crt;
VAR
  x0,y0,x1,y1 :Byte;
BEGIN
  ...
  x0:=WindMinX;  x1:=WindMaxX;
  y0:=WindMinY;  y1:=WindMaxY;
  Window(25,1,32,2);
  ...
  Window(x0,y0,x1,y1);
  ...
END.
```

Moving in the Window

The cursor can be placed within the window by the GotoXY procedure. The position (1,1) is the upper left corner of the window. If either the specified x or y coordinate is outside of the current window, the cursor position will not be changed.

All reading and writing to the display are made starting at the cursor position.

The current cursor position can be read by the WhereX and WhereY functions. They can be used to restore the cursor position after writing text to the display.

Example:

Use the line edit function of the Read procedure to write editable text for that procedure (line edit is described on page 36), and read the edited line back to the same variable.

```
USES Crt;
VAR
  s    :STRING;
  xs,ys:Byte;
BEGIN
  ...
  xs:=WhereX;  ys:=WhereY;
  Write(s);
  GotoXY(xs,ys);
  ReadLn(s);
  ...
END.
```

Chapter 30

Graphics

The display of the PB-2000 is capable of displaying graphics. Every *pixel* in the display can be controlled. In the Crt unit, procedures Plot, Line, and Point are used for this purpose.

These procedures do not directly update the LCD screen to the contents of the graphics buffer. Instead, the Display procedure is needed to actually display the graphic data written. This solution has been used to speed the writing of graphic data. Since most graphic figures consists of more than a few lines, all lines for the figure can be generated before the complete graphic figure is shown on the display.

All coordinates for graphic data are numbered from zero upwards, with (0,0) at the upper left corner of the display. For graphic data, the window facility is not used. All coordinates are according to the display.

Text written to the display is converted to graphic data, so any text on the display will behave as graphic pixels. When the program ends and goes into **CAL** mode, only the character buffer area will be used to move data into the **CAL** buffer. All graphic data will then disappear.

Plot Pixels

Separate pixels are set and reset by the Plot procedure. In addition to the coordinates, the pixel color is entered. A color value of 0 will set the pixel off, value 1 will set it on, and value 2 will invert the previous value of the pixel (using XOR).

Coordinates outside of the display can be entered to this procedure. In that case, nothing will be changed in the display.

Example:

Invert the pixels for the upper left character (6 pixels in x, and 8 in y direction).

```
USES Crt
VAR
  x,y :Integer;
BEGIN
  ...
  FOR y:=0 TO 7 DO
   FOR x:=0 TO 5 DO
    Plot(x,y,2);
  Display;
  ...
END.
```

Read Pixels

The current pixel value can be read by the Point function. The returned value will be 0 if the pixel is off, and 1 if it is on. If the coordinate is outside of the display, the value 0 is returned.

Example:

This procedure will check a pixel, and set two other pixels (in backward x and y direction) if the pixel is set. Last, the checked pixel will be cleared.

```
PROCEDURE CheckPixel (x,y:Integer);
  BEGIN
    IF Point(x,y)=1 THEN BEGIN
      Plot(x-1,y,1);
      Plot(x,y-1,1);
    END;
    Plot(x,y,0);
  END;
```

Draw Lines

In addition to setting single pixels, lines can be drawn by the Line procedure. As with the Plot procedure, a color parameter is entered. Value 0 will clear all pixels in the line, value 1 will set all pixels in the line, and value 2 will invert all pixels in the line.

The starting and/or ending coordinates of the line can be set outside of the display. In order to make an exact line part inside of the display, the line routine will actually try to write all pixels on the line. For very long lines, this can take some time. If this exact drawing is not needed, the line should be manually clipped at a position nearer to the display border.

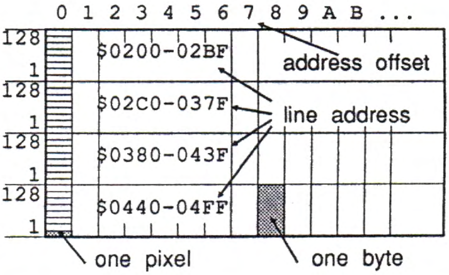
Example:

Fill the display with a striped pattern (every other line set).

```
PROGRAM LineDemo;
USES Crt;
VAR
  y :Integer;
  ch:Char;
BEGIN
  FOR y:=0 TO 15 DO
   Line(0,y*2,191,y*2,1);
  Display;
  ch:=ReadKey;
END.
```

Direct Graphic Access

When fast writing of graphic data is needed, the graphic display buffer can be directly accessed. This buffer is located at addresses between $0200 and $04FF, in the way that can be seen in the figure to the right. The upper left pixel is located at address $0200, in the topmost bit (value 128). To use the graphic buffer, a variable should be set absolute to the graphic buffer address, such as:


0 1 2 3 4 5 6 7 8 9 A B ...
128
1
128
1
128
1
128
1
$0200-02BF
$02C0-037F
$0380-043F
$0440-04FF
address offset
line address
one pixel
one byte

```
graphBuffer:ARRAY [0..767] OF Char ABSOLUTE $0200;
```

When the graphic area has been written to directly, the UpdateRegion procedure is needed before the Display procedure is called. This procedure has to be called to signal which part of the graphic area has been changed.

Example:

This program excerpt will display a checkered pattern in the graphic buffer and on the display. The graphic buffer is here accessed as a Word array, to set 16 pixels at one time.

```
USES Crt;
CONST
  GraphBufLen=192*2-1;
VAR
  graphBuffer :ARRAY [0..GraphBufLen] OF Word
               ABSOLUTE $0200;
  p :Word;
BEGIN
  ...
  FOR p:=0 TO GraphBufLen DO
   graphBuffer[p]:=$55AA;
  UpdateRegion(0,0,191,31);
  Display;
  ...
END.
```

Chapter 31

System Unit

All identifiers in the System unit are automatically visible to all programs and units. In this unit, identifiers declared in the DL-Pascal language are defined. This includes the normal Pascal identifiers.

System Types

The predefined types available in a Pascal program are all declared in the System unit. They are:

`ShortInt`	Value between –128 and 127, using 1 byte.
`Byte`	Value between 0 and 255, using 1 byte.
`Integer`	Value between –32768 and 32767, using 2 bytes.
`Word`	Value between 0 and 65535, using 2 bytes.
`LongInt`	Value between –2147483648 and 2147483647, using 4 bytes.
`Real`	Value between –9.99E99 and –1.E–99, 1.E–99 and 9.99E99, and the value 0, using 8 bytes.
`Boolean`	The values False and True, using 1 byte.
`Char`	Characters between #0 and #255, using 1 byte.
`Pointer`	Near pointer (in normal RAM), using 2 bytes.
`LongPtr`	Far pointer with segment and offset, using 4 bytes.
`Text`	Text file type, using 96 bytes.

System Variables

There are also some variables defined in the System unit. They are special variables used by some of the procedures and functions in this unit.

angle

Declaration:	`angle :Byte`
Description:	The current trigonometric functions angle unit. The value can be 0 for degrees, 1 for radians, and 2 for grads. The default setting is 1 (radians) when the program is started.

Example:

```
angle:=0;
WriteLn('sin(pi/4)=',SIN(45):255);
```

errorAddr

Declaration:	`errorAddr :Pointer`
Description:	When an error occurs in the program, the code address is set in this variable. For normal termination, it is set to NIL.
Example:	see chapter 28.

exitCode

Declaration:	`exitCode :Word`
Description:	A program terminated by Halt or RunError, or by a run-time error, sets this variable to the error code. The lower byte of this variable is used by the system to be set in the ERR variable in the **CAL** mode.
Example:	see chapter 28.

exitProc

Declaration:	`exitProc :PROCEDURE`
Description:	A terminating program will call the current exitProc procedure. Normally, this procedure should restore the earlier exitProc

procedure. The default procedure is the System unit's termination procedure, which writes an error message depending on the values in exitCode and errorAddr.

Example: see chapter 28.

heapError

Declaration: `heapError :FUNCTION (size:Word) :Byte`

Description: This function variable is called when a call to GetMem (or New) cannot be executed due to too little space on the heap. The function can then remove some unneeded data on the heap and return with a code. This code can be 0 to generate a run-time error, 1 to return to the caller of the GetMem or New procedure with the pointer set to NIL, and any other value to retry the allocation.

Example: see chapter 26.

input

Declaration: `input :Text`

Description: The standard Pascal input file. The input is normally from the keyboard, but by using redirection can be from any file or device.

Example: see chapter 27.

output

Declaration: `output :Text`

Description: The standard Pascal output file. This output is normally to the display, but can be redirected to any file or device.

Example: see chapter 27.

System Procedures and Functions

The following is a description of all procedures and functions defined in the System unit. Some of these are decoded internally, and as such cannot be used by the address operators. This is noted at those procedures and functions.

Abs

Declaration:

```
Abs (x:int) :int
Abs (x:Real) :Real
```

Description: Calculate the absolute value of the integer or real parameter. The return type is the same as the parameter type. For Byte and Word parameters, no code will be generated.

This function cannot be used with the address operator. It can be used in a constant expression.

Example:

```
IF x<0 THEN Write(Sqrt(Abs(x)):255,’i’);
```

Addr

Declaration:

```
Addr (x) :LongPtr
```

Description: Return the address to the variable x. The function is identical to the address operator &.

The function cannot be used with another address operator.

Example:

```
VAR p :^Integer;
    i1:Integer;
BEGIN
   ...
   p:=Addr(i1);
   ...
END.
```

Append

Declaration:

```
Append (VAR t:Text)
```

Description: Open the text file t in append mode. The write pointer will be placed at the end of the file, so further writing is appended at the end of the file.

The variable t must have been assigned a file specifier using the Assign procedure.

It is an error if the specified file does not exist.

This procedure cannot be used with the address operators.

Example: Append some text to the end of the text file **TEXT1**.

```
VAR
  t :Text;
BEGIN
  Assign(t,'TEXT1');
  Append(t);
  WriteLn(t,'Appended text.');
  Close(t);
END.
```

ArcCos

Declaration: **`ArcCos (x:Real) :Real`**

Description: Calculate the arccosine of the real type argument.

Example: `x:=ArcCos(0.15);`

ArcCosH

Declaration: **`ArcCosH (x:Real) :Real`**

Description: Calculate the hyperbolic arccosine of the real type argument.

Example: `res:=ArcCosH(x*0.1);`

ArcSin

Declaration: **`ArcSin (x:Real) :Real`**

Description: Calculate the arcsine of the real type argument.

Example: `x:=ArcSin(inVal);`

ArcSinH

Declaration: **`ArcSinH (x:Real) :Real`**

Description: Calculate the hyperbolic arcsine of the real type argument.

Example: `x2:=ArcSinH(1.0);`

ArcTan

Declaration: **`ArcTan (x:Real) :Real`**

Description: Calculate the arctangent of the real type argument.

Example: `x:=ArcTan(1.0);`

ArcTanH

Declaration: **`ArcTanH (x:Real) :Real`**

Description: Calculate the hyperbolic arctangent of the real type argument.

Example: `WriteLn(ArcTanh(val1));`

Assign

Declaration:

```
Assign (VAR f; name:STRING)
Assign (VAR f; name:STRING; flags:Byte)
```

Description: Assign a file specifier to a file or text variable. For a text variable, the default file type is S, and a typed or untyped file has the default file type T. The file type is only used if the Rewrite procedure is used.

An optional parameter can be given, to keep some parts of a previously assigned file specifier in variable f. If not entered, a value of zero is used. To use this parameter, sum the needed function values, one from each field.

The first field is used to specify the device. The value 0 specifies the RAM device, if no device is set in the file specifier. A value of 1 will not change the previous device, if no device is set in the new file specifier.

The second field is used with the file name. A value of 0 is used if the file name is always to be set. Value 2 specifies that the old file name is not changed if no name is entered in the new file specifier. The value 4 is used for normal file name setting if a name is entered, else the name is set to all wild cards (to match any file). The last value 6 works like value 2, except for the positions in the new file where a wild card character is found (? or *). In those positions, the old file name will remain.

The third field controls the extension. The function of each value is identical to the file name field, except that it controls the extension. Value 0 always sets the extension. Value 8 does not change the old extension if no new extension is entered. Value 16 will set the extension either to the new one or to all wild cards. Last, the value 24 will keep the old characters where a wild card is used in the new name, or not change the extension if none has been entered.

The last field controls the file type. The possible values differ for text and file type variables being assigned. The values within parentheses specify the values used with file type variables, and the other is for text type variables.

A value of 0 (128) will match any file type, and use the file type S if Rewrite is used. The value 32 (32) is used if the old file type is not to be changed if no new file type is entered. Value 64 (196) sets the default file type to **S** (sequential file). Last, the value 128 (0) sets the default file type to **T** (typed file).

The Assign procedure cannot be used with the address operators.

Example: see chapter 27.

BlockRead

Declaration:

```
BlockRead (VAR f:File; VAR buf; cnt:Word)
BlockRead (VAR f:File; VAR buf; cnt:Word;
                              VAR actRead:Word)
```

Description: This procedure is used to read a data block from an untyped file. The file variable must have been assigned and the file opened, before calling the procedure.

A maximum of cnt blocks will be read from the file to the buffer variable buf. The block size is specified in the Reset procedure. The actually read number of blocks can be returned in the optional parameter actRead. This parameter returns the number of fully read blocks, not any partially read ones.

The buffer variable should be large enough to hold the specified number of blocks.

This procedure cannot be used with the address operators.

Example: see chapter 27.

BlockWrite

Declaration:

```
BlockWrite(VAR f:File; VAR buf; cnt:Word)
BlockWrite(VAR f:File; VAR buf; cnt:Word;
                               VAR actWrite:Word)
```

Description: This procedure is used to write a block to an untyped file. The file variable must have been assigned and the file opened, before calling the procedure.

Cnt blocks will be written to the file from the buffer variable buf. The block size is specified in the Rewrite procedure. The actually written number of blocks can be returned in the optional parameter actWrite. This variable specifies the number of fully written blocks, not any partially written ones.

This procedure cannot be used with the address operators.

Example: see chapter 27.

Ceil

Declaration: `Ceil (x:Real) :Real`

Description: Calculate the smallest integer number larger than or equal to the parameter value (round the parameter value up).

Example:

```
roundedUp:=Ceil(4.5);
```

Chr

Declaration: `Chr (x:Byte) :Char`

Description: Convert the integer parameter value to a Char type value. This function can be used in a constant expression. It cannot be used with an address operator.

Example: To convert a decimal digit to the corresponding character, the following expression can be used:

```
charVal:=Chr(digit+48);
```

Close

Declaration: `Close (VAR f)`

Description: Close the file connected to the file or text variable f. The file must have been previously opened by the Reset or Rewrite procedure.

This procedure cannot be used with the address operators.

Example: see chapter 27.

Concat

Declaration: `Concat (s1,s2 ... :STRING) :STRING`

Description: Concatenate the string type parameters together, and return the concatenated string. This function is used in the same way as the concatenate operator +.

The function cannot be used with an address operator.

Example: The following program part will search for the string **CASIO** in string variable s, and if found remove it from the string and place it in another string newS.

```
VAR
  p :Word;
  s,newS :STRING[80];
BEGIN
  ...
  p:=Pos('CASIO',s);
  IF p>0 THEN BEGIN
    newS:=Copy(s,p,5);
    Delete(s,p,5);
    newS:=Concat('String ',newS,
                 ' found.');
  END;
  ...
END.
```

Copy

Declaration:	`Copy (s:STRING; index,count:Byte) :STRING`
Description:	Return a substring of the parameter s, from the position index and a maximum of count bytes forward. The first character in the string has index 1. This function cannot be used with the address operators.
Example:	see Concat.

Cos

Declaration:	`Cos (x:Real) :Real`
Description:	Calculate the cosine of the real type parameter.
Example:	`x:=Cos(2.5);`

CosH

Declaration:	`CosH (x:Real) :Real`
Description:	Calculate the hyperbolic cosine of the real type parameter.
Example:	`res:=CosH(x);`

Dec

Declaration:	`Dec (VAR v)` `Dec (VAR v; count:Integer)`
Description:	Decrement the ordinal type variable v by one or by the specified count. This procedure cannot be used with the address operators.

Example: In the string s, scan from the high end until a space is found (or all of the string is searched).

```
VAR
  p :Byte;
  s :STRING;
BEGIN
  ...
  p:=Length(s);
  WHILE (p>0) AND (s[p]<>' ') DO
   Dec(p);
  ...
END.
```

Delete

Declaration: **`Delete (VAR s:STRING; index,count:Byte)`**

Description: Delete a part of the string in s, starting at index and a maximum of count characters forward. The first character in the string has index value 1.

This procedure cannot be used with an address operator.

Example: see Concat.

Dispose

Declaration: **`Dispose (VAR p:Pointer)`**

Description: Free the heap area allocated to the pointer type parameter. This pointer must previously have been assigned a value by the New procedure.

This procedure is identical to calling GetMem with the size of the variable pointed to:

```
GetMem (p, SizeOf(p^));
```

The procedure cannot be used with the address operators.

Example: see chapter 26.

Eof

Declaration:	`Eof :Boolean` `Eof (VAR f) :Boolean`
Description:	Return the current end-of-file status of the specified text or file type variable. If no parameter is entered, the standard text file input is used. In any case, the variable must have been assigned a file specifier, and the file opened by Reset or Rewrite. For a typed or untyped file, this function returns True if the file pointer is placed at the very end of the file. With a text file, this function returns True if the next character to be read is the end-of-file character (ordinal value 26). This function cannot be used with the address operators.
Example:	see chapter 27.

Eoln

Declaration:	`Eoln :Boolean` `Eoln (VAR t:Text) :Boolean`
Description:	Return the current end-of-line status of the text file t. If no parameter is entered, the standard text file input is used. The file must have been opened by Reset or Rewrite before calling this function. The function returns True if the next character to be read is the carriage-return character (ordinal value 13). This function cannot be used with an address operator.
Example:	see chapter 27.

Erase

Declaration:	`Erase (VAR f)`
Description:	Delete the file connected to the text or file variable f. The file specifier must have been assigned by the Assign procedure, and the file must currently not be opened. This procedure cannot be used with an address operator.

Example: A new file is created, and the previous file with that name is set as a backup file. The old backup file is deleted.

```
VAR
  f :FILE;
  r :Byte
BEGIN {$I-}
  Assign(f,'FILE.BAK');
  Erase(f);
  r:=IOResult;
  Assign(f,'FILE');
  Rename(f,'FILE.BAK');
  r:=IOResult;
  Assign(f,'FILE');
  ...
END.
```

Exit

Declaration: **`Exit`**

Description: Return from the current function or procedure. This procedure can be used anywhere within the body of the subroutine. It can also be used in the program's main part, in which case it works like the Halt procedure without any parameter.

This procedure cannot be used with an address operator.

Example: This function scans a string until a space is found, and returns the position of the space. If none is found then zero is returned.

```
FUNCTION ScanSpace (s:STRING) :Byte;
  VAR p:Byte;
  BEGIN
    FOR p:=1 TO Length(s) DO
     IF s[p]=' ' THEN BEGIN
       ScanSpace:=p;
       Exit;
     END;
    ScanSpace:=0;
  END;
```

Exp

Declaration: **`Exp (x:Real) :Real`**

Description: Calculate the exponential value of the real type parameter (e^x).

Example: `exp1:=Exp(1.0);`

FilePos

Declaration: `FilePos (VAR f:File) :LongInt`

Description: Return the current file pointer position. This position is expressed in blocks, where the size of each block is specified when Reset or Rewrite is called. The file variable must have been assigned a file specifier, and the file opened before using this function.

This function cannot be used with the address operators.

Example: see chapter 27.

FileSize

Declaration: `FileSize (VAR f:File) :LongInt`

Description: Return the current file size. The size is expressed in blocks, where each block has the size specified at Reset or Rewrite. The file variable must have been assigned a file specifier, and the file opened before using this function.

This function cannot be used with the address operators.

Example: see chapter 27.

FillChar

Declaration: `FillChar (VAR x; count:Word; val)`

Description: Set the contents of variable x to the ordinal value val. The size to fill is specified in bytes by count.

Variable x may also be a variable defined in another bank than bank 1 (normal RAM memory).

This procedure cannot be used with an address operator.

Example: Fill the string variable s with 32 question marks, and also set the length byte to 32.

```
VAR
  s :STRING;
BEGIN
  ...
  FillChar(s[1],32,'?');
  s[0]:=Chr(32);
  ...
END.
```

Floor

Declaration: **`Floor (x:Real) :Real`**

Description: Return the largest integer value smaller than or equal to the parameter value (round the value down).

Example: `roundedDown:=Floor(4.5);`

Frac

Declaration: **`Frac (x:Real) :Real`**

Description: Return the fractional part of the real type parameter.

Example: `fracPart:=Frac(4.5);`

FreeMem

Declaration: **`FreeMem (VAR p:Pointer; size:Word)`**

Description: Release the memory allocated on the heap for the pointer variable p. This area must previously have been allocated by the GetMem procedure. The same size as when GetMem was called must be used.

Example: see chapter 26.

GetMem

Declaration: **`GetMem (VAR p:Pointer; size:Word)`**

Description: Allocate memory on the heap for the pointer variable p. The variable size specifies the number of bytes to allocate.

If not enough memory is available on the heap, the heapError variable procedure is called; normally generating a run-time error.

Example: see chapter 26.

Halt

Declaration: **`Halt`**
`Halt (errorCode:Word)`

Description: This procedure is used to stop execution of the program without writing an error message. An error code parameter can be used, which can later be read in the CAL mode by the ERR variable. If no parameter is entered, the error code is set to zero.

This procedure cannot be used with an address operator.

Example: see chapter 28.

Hi

Declaration: **`Hi (x:Word) :Byte`**

Description: Return the high byte of the Word type parameter.

This function can be used in constant expressions. It cannot be used with an address operator.

Example: Extract the high and low bytes of variable w, and finally swap the high and low bytes in w.

```
VAR
  w :Word;
  wLo,wHi :Byte;
BEGIN
  ...
  wLo:=Lo(w);
  wHi:=Hi(w);
  w:=Swap(w);
  ...
END.
```

Inc

Declaration: **`Inc (VAR v)`**
`Inc (VAR v; count:Integer)`

Description: Increment the ordinal type variable v by one, or by count. This procedure cannot be used with an address operator.

Example: Check the array arr from the beginning to the first value larger than 20. Only each other position is searched.

```
VAR
  arr:ARRAY [0..255] OF Byte;
  p  :Word;
BEGIN
  ...
  p:=0;
  WHILE (arr[p]<=20) DO
   INC(p,2);
  ...
END.
```

Insert

Declaration: **`Insert (s:STRING; VAR d:STRING; ind:Byte)`**

Description: Insert the string s into the string dst. The length of string variable d can be of any size. The insert position is specified by the index position in ind, where the first character in the string is at position 1.

If the new string cannot fit in the size specified for d, the resulting string is truncated.

This procedure cannot be used with the address operators.

Example: Search for a string starting with **DL** in s, and insert the resulting string to the beginning of string inS.

```
VAR
  p :Word;
  s,inS :STRING[120];
BEGIN
  ...
  p:=Pos('DL',s);
  Insert(Copy(s,p,255),inS,1);
  ...
END.
```

Int

Declaration: **`Int (x:Real) :Real`**

Description: Return the integer part of the real type parameter value (for positive values round down, and for negative values round up).

Example: `rounded:=Int(4.5);`

IOResult

Declaration: **`IOResult :Word`**

Description: Return the last error code, resulting from a file operation. After this function has been called, the internal error code is reset.

For this function to be of any value, the compiler option {$I–} must have been specified. If this is not done, any error from a file operation immediately terminates the execution of the program.

This function cannot be used with the address operators.

Example: see chapter 27.

Length

Declaration: **`Length (s:STRING) :Byte`**

Description: Return the current length of the string. The length of the string is stored at the zeroth character in the string.

This function can be used in a constant expression, if the parameter is a constant string or a character. The function cannot be used with an address operator.

Example: see Dec.

Ln

Declaration: **`Ln (x:Real) :Real`**

Description: Return the natural logarithm of the real type parameter.

Example: `lnVal:=Ln(x);`

Lo

Declaration: `Lo (x:Word) :Byte`

Description: Return the low byte of the Word size parameter. This function can be used in a constant expression, but cannot be used with an address operator.

Example: see Hi.

Log

Declaration: `Log (x:Real) :Real`

Description: Calculate the common logarithm of the real type parameter.

Example:

```
logVal:=Log(x);
```

Mark

Declaration: `Mark (VAR p:Pointer)`

Description: Store the current end of the allocated memory in the heap to the parameter p.

Example: see chapter 26.

Max

Declaration: `Max (t:Type) :Type`

Description: Return the highest value possible with the ordinal type t. The returned type is the same as the parameter type.

This function is allowed in a constant expression, and will always return a constant value. The function is not allowed with an address operator.

Example: The array oa is initialized to random real values. The index of that array is an enumerated type, and Min and Max are used to get the minimum and maximum values of the array index, regardless of any future changes to the enumerated type.

```
TYPE
  Color=(red,green,blue,gray);
VAR
  oa :ARRAY [Color] OF Real;
  ip :Color;
BEGIN
  Randomize;
  FOR ip:=Min(Color) TO Max(Color) DO
   oa[ip]:=Random;
  ...
END.
```

MaxAvail

Declaration: **`MaxAvail :Word`**

Description: Return the largest possible block size that can be allocated on the heap. It is either one block freed by FreeMem, or the unallocated block on the top of the heap.

Example: See chapter 26.

MemAvail

Declaration: **`MemAvail :Word`**

Description: Return the total size of free blocks on the heap, including the unallocated part on the top of the heap.

Example: To write the current free heap size for dynamic memory, the following statement can be used:

```
WriteLn('Free heap=',MemAvail);
```

Min

Declaration: **`Min (t:Type) :Type`**

Description: Return the smallest value possible with the ordinal type t. The returned type is the same as the parameter type.

This function is allowed in a constant expression, and will always return a constant value. The function is not allowed with an address operator.

Example: see Max.

Move

Declaration: **`Move (VAR source,dest; count:Word)`**

Description: Copy the memory contents from source to dest. The number of bytes to copy is specified by count. The copy is always made in a non-overlapping direction.

The parameters source and dest may also be variables defined in a bank other than bank 1.

This procedure cannot be used with the address operators.

Example: Move the first ten bytes from the record variable rec to the string s.

```
VAR
  s  :STRING[99];
  rec:RECORD
        len :Byte;
        data:ARRAY [1..99] OF Char;
      END;
BEGIN
  ...
  Move(rec,s,10);
  ...
END.
```

New

Declaration: **`New (VAR p:Pointer)`**

Description: Allocate space for the variable pointed to by the pointer p on the heap. The allocated variable is then accessed by p^.

If there is not enough room on the heap for the variable, the heapError procedure variable is called. This normally generates a run-time error.

This procedure works in the same way as if GetMem had been called with the variable size:

```
GetMem (p, SizeOf(p^))
```

The New procedure cannot be used with an address operator.

Example: see chapter 26.

Odd

Declaration: `Odd (l:LongInt) :Boolean`

Description: Return True if the integer parameter has an odd value (1,3,...). This function can be used in a constant expression, but cannot be used with the address operator.

Example: Check the random number generator, by generating a value of 0 or 1 and counting the number of 1's generated.

```
VAR
   count:Word;
   loop :Word;
BEGIN
   Randomize;
   count:=0;
   FOR loop:=1 TO 100 DO
    IF Odd(Random(2)) THEN Inc(count);
   WriteLn('Of 100 times: ',count);
END.
```

Ofs

Declaration: `Ofs (x) :Word`

Description: Return the offset address to the variable x (the address within a bank).

This function cannot be used with an address operator.

Example: This procedure writes the address to the parameter set when calling the procedure, in hexadecimal.

```
PROCEDURE WrAddr (p :LongPtr);
   BEGIN
      WriteLn('Seg=',%LongInt(Seg(p^)):4,
              ',Ofs=',%LongInt(Ofs(p^)):4);
   END;
```

Ord

Declaration: `Ord (x) :int`

Description: Return the ordinal value of the ordinal type variable x. The type of the result is the integer type corresponding to the size of the parameter (e.g. a Char will return a Byte type).

This function can be used in a constant expression. No code is actually generated by this function, and therefore this function cannot be used with an address operator.

Example: Convert a character digit to the numeric digit value.

```
numVal:=Ord(charDigit)-48;
```

Page

Declaration:

```
Page
Page (VAR t:Text)
```

Description: Write a page character to the text file specified, or to the standard output file. This character is the form-feed character (#12).

This procedure cannot be used with the address operator.

Example: After writing a text to a file (to be printed), write a form-feed character to move the page out of the printer.

```
BEGIN
   ...
   WriteLn('Last line on the page.');
   Page;
   ...
END.
```

ParamCount

Declaration: **`ParamCount :Byte`**

Description: Return the entered number of parameters, when calling the program from the command line in **CAL** mode. When the program is called directly from the **MENU** mode, or from a preset file in **CAL** mode, this function returns the value zero.

Example: If the program was called from the **CAL** mode with a parameter, then use that parameter as a text file specifier. In other cases, read the specifier from the keyboard.

```
VAR
  s :STRING;
  t :Text;
BEGIN
  IF ParamCount>0 THEN s:=ParamStr(1)
  ELSE BEGIN
    Write('Enter file name:');
    ReadLn(s);
  END;
  Assign(t,s);
  ...
END.
```

ParamStr

Declaration: **`ParamStr (num:Byte) :STRING`**

Description: Return the parameter num, as entered on the command line. Only parameters of number 0 through 9 can be located. The first entered parameter has number 1. Parameter zero is the program name, as entered on the command line.

When the program is called from the **MENU** mode, or from a preset file in **CAL** mode, an empty string is returned for all parameters. This can be used to detect where the program was called from. If the string for parameter zero (the program name) is empty, the file was called from **MENU** or preset mode. If something is returned for parameter zero, the program was called from the command line.

Example: see ParamCount.

Pi

Declaration: **`Pi :Real`**

Description: Return the internal approximate value for pi.

Example: `WriteLn('PI=',pi:255);`

Pos

Declaration: **`Pos (subStr,s:STRING) :Byte`**
`Pos (subStr,s:STRING; ind:Byte) :Byte`

Description: Search the string s for any occurrence of the substring subStr. An optional parameter ind can be used to specify the starting search position. If this is not entered, the search starts from the beginning of string s.

Returned from the function is the index position of the found string, or the value zero if the substring is not found. The position of the first character in the string has value 1.

This function cannot be used with an address operator.

Example: see Concat and Insert.

Power

Declaration: **`Power (x,y:Real) :Real`**

Description: Return the result of the operation x^y. Both x and y are real numbers. This function is calculated using the formula

$$10^{(y \log(x))}$$

and therefore the value of x must be larger than zero.

Example:

```
thirdRoot:=Power(x,1/3);
```

PowerInt

Declaration: **`PowerInt (x:Real; y:Integer) :Real`**

Description: Return the result of the operation x^y, where the exponent y is an integer value. This function is calculated using mathematical expansion of the expression, and will return a more accurate value than using Power. In addition, negative numbers can be used for parameter x.

Example:

```
cube:=PowerInt(-2.0,3);
```

Pred

Declaration: **`Pred (x:Type) :Type`**

Description: Return the preceding ordinal number of the parameter type. This is the value x–1. The returned type is the same as the parameter type.

This function can be used in constant expressions. It cannot be used with the address operator.

Example: Find the position in string s of a space, preceded by a colon, and where the next character is not a space.

```
VAR
  p :Word;
  s :STRING
BEGIN
  ...
  s:='Find only: this';
  FOR p:=2 TO Length(s)-1 DO
   IF s[p]=' ' THEN
    IF (s[pred(p)]=':') AND
       (s[succ(p)]<>' ' THEN
     Write(p);
  ...
END.
```

Ptr

Declaration: **`Ptr (seg,ofs:Word) :LongPtr`**

Description: Create a far pointer from the segment (bank) part seg and the offset ofs. This pointer will then point to the specified address.

This function cannot be used with address operators.

Example: see Ofs.

Random

Declaration: **`Random :Real`**
`Random (x:Word) :Word`

Description: Return a random number, built on a pseudo-random generator. If no parameter is entered, the returned number is a positive real value larger than 0 but smaller than 1.

If a parameter is used, the returned number is in the range 0 to x–1 (i.e. it has x distinct values).

The procedure Randomize should be used in the beginning of the program to initialize the random number generator with a random value.

This function cannot be used with the address operator.

Example: see Max and Odd.

Randomize

Declaration: `Randomize`

Description: Initialize the random number generator with a random number. If this procedure is not used, the Random function will return the same "random" number sequence each time the program is run.

Example: see Max and Odd.

Read (typed file)

Declaration: `Read (VAR f:File; VAR v1; ...)`

Description: The Read procedure is used in a typed file f to read one or more variables from that file. The type of the variables must be the same as the element type of the file type.

This procedure cannot be used with an address operator.

Example: see chapter 27.

Read, ReadLn (text file)

Declaration:

```
Read (VAR v1; ...)
Read (VAR t:Text; VAR v1; ...)
```

Description: This procedure is used to read data from a text file t, or from the standard input text file. The variables can be of integer, real, character, and string types.

The difference between Read and ReadLn is that after reading the variables, the ReadLn procedure will skip any unread data until the end of the line (including the new-line character). When Read is used, the remainder of the line will be kept for the next read from this file.

For a character type variable, the next character in the file will be read. At the end of the line, the carriage-return character (#13) will be read, followed by the new-line character (#10).

A string variable will read the reminder of the line, up to the end-of-line character. If the number of characters read is too large for the string, the read string is truncated. Note that the end-of-line character is not read. For successive reading of string variables, separate ReadLn calls should be made for each string, so the intermediate end-of-lines will be removed.

When reading integer and real variables, data from the current text file position will be read, up to the next space or control character (such as the new-line character). The data is then converted to an integer or real type. If a non-allowed character is found, nothing will be assigned to the variable, but execution will terminate with an error message (or an error code will be set in the IOResult variable if compiler option {$I–} is set).

A LongInt variable can be read in hexadecimal format by preceding the variable by a percent sign.

This procedure cannot be used with an address operator.

Example: see chapter 27.

RealError

Declaration: **`RealError (x:Real) :Byte`**

Description: Return the error code set in the real parameter x. This function can be used after an expression assigning a value to the variable x. The error code will be the same as can be seen in appendix E, or zero if no calculation error was existing.

The compiler option {$F–} must be set before the expression, or a real evaluation error will automatically terminate execution of the program.

Example: The following program part will check the result of the expression, and write a message if an error was detected.

```
VAR
  r,cr,res :Real;
BEGIN {$F-}
  ...
  res:=Sqrt(r)/cr;
  IF RealError(res)>0 THEN
   WriteLn('Expression error.');
  ...
END.
```

Release

Declaration: **`Release (VAR p:Pointer)`**

Description: This procedure is used in combination with the Mark procedure, to restore the end of the heap position.

It should not be used in combination with FreeMem (or Dispose), as this procedure will clear the free list structure. Any free areas within the allocated heap space will then be lost.

Example: see chapter 26.

Rename

Declaration: `Rename (VAR f; newName:STRING)`

Description: Rename the file connected to the variable f. A file specifier must have been assigned to the variable, and the file must not be opened. After renaming the file, the new name and extension will also be set in the file variable.

The new file name and extension can use wild card characters (* and ?) to keep some (or all) characters of the old file name and extension.

An error is generated if the file does not exist, or a file using the new name already exists.

This procedure cannot be used with an address operator.

Example: see Erase.

Reset

Declaration:
```
Reset (VAR f)
Reset (VAR f:File; recSize:Word)
```

Description: Open the file connected to the file variable f, or open the text variable f for input. A file specifier must have been assigned to the variable by the Assign procedure. It is an error if the file does not exist.

An untyped file can have an optional parameter specifying the block size. If omitted, a standard value of 128 is used. The block size should be set to 1 to be able to read every byte of the file. With another size, the last block in the file cannot be read if the size of that block is not exactly the block size.

This procedure cannot be used with an address operator.

Example: see chapter 27.

Rewrite

Declaration: **`Rewrite (VAR f)`**
`Rewrite (VAR f:File; recSize:Word)`

Description: Create the file connected to the file or text variable f. A text file is then opened for output. A file specifier must have been assigned to the file variable by the Assign procedure. If a file with this name already exists, that file is deleted and a new file is created.

An untyped file can have an optional parameter specifying the block size. If omitted, a standard value of 128 is used. The block size should be set to 1 to be able to write data down to a single byte in the file. The data written must be an exact multiple of the block size.

This procedure cannot be used with an address operator.

Example: see chapter 27.

Round

Declaration: **`Round (x:Real) :LongInt`**

Description: Convert the real type parameter to a LongInt value, by rounding the real value. No error will be generated if the real value is too large.

This function can be used in constant expressions. It cannot be used with the address operator.

Example:

```
intVar:=Round(Pi);
```

RoundTo

Declaration: **`RoundTo (x:Real; deci:ShortInt) :Real`**

Description: Round the real value x to the specified number of decimals. If the number of decimals is negative, the rounding position will be to the left of the decimal point.

Example: The following program will write the entered value rounded to different number of decimals.

```
PROGRAM RoundToDemo;
VAR
  r   :Real;
  deci:ShortInt;
BEGIN
  Write('Enter number:');
  ReadLn(r);
  FOR deci:=5 DOWNTO -5 DO
   WriteLn(deci:2,'->',
                RoundTo(r,deci):255);
END.
```

RoundDigit

Declaration: **`RoundDigit (x:Real; digs:ShortInt) :Real`**

Description: Round the real value `x` to the specified number of significant digits. This rounding is done regardless of the decimal point position.

Example: The following program will write the entered value rounded to different number of digits.

```
PROGRAM RoundDigitDemo;
VAR
  r   :Real;
  dig:ShortInt;
BEGIN
  Write('Enter number:');
  ReadLn(r);
  FOR dig:=1 TO 12 DO
   WriteLn(dig:2,'->',
                RoundDigit(r,dig):255);
END.
```

RunError

Declaration: **`RunError`**
`RunError (errorCode:Word)`

Description: Stop execution of the program, and set the error code to the specified parameter value, or zero if no parameter is entered. In addition, an error message will be written by the exit procedure (the errorAddr variable will be set).

This procedure cannot be used with the address operator.

Example: see chapter 28.

Seek

Declaration: `Seek (VAR f:File; pos:LongInt)`

Description: Set the file position for a typed or untyped file f at block position pos. The actual file position in bytes will be pos * block size from the beginning of the file. The file variable must have been assigned a file specifier, and the file must have been opened.

This procedure cannot be used with an address operator.

Example: see chapter 27.

SeekEof

Declaration:

```
SeekEof :Boolean
SeekEof (VAR t:Text) :Boolean
```

Description: This function works in the same way as the Eof function, except that any spaces or control characters are first skipped before testing the end-of-file status.

This function cannot be used with an address operator.

Example: see chapter 27.

SeekEoln

Declaration:

```
SeekEoln :Boolean
SeekEoln (VAR t:Text) :Boolean
```

Description: This function works in the same way as the Eoln function, except that any spaces or control characters (except the end-of-line character) are skipped before testing the end-of-file status.

This function cannot be used with an address operator.

Example: see chapter 27.

Seg

Declaration: **`Seg (x) :Word`**

Description: Return the segment (or bank) part of the variable x. For a normal variable in RAM, the bank number is 1.

This function can be used in a constant expression, if the variable is a normal RAM variable. In this case, the function will return the constant value 1.

This function cannot be used with the address operators.

Example: see Ofs.

SetTextBuf

Declaration:

```
SetTextBuf (VAR t:Text; VAR buf)
SetTextBuf (VAR t:Text; VAR buf; sz:Word)
```

Description: Set another buffer for text file variable t. The new buffer will be set at the variable buf, and the size will be of sz bytes. If the size parameter is not used, the buffer size will default to 64 bytes. A larger buffer will speed reading from or writing to the text file, as more data can be buffered.

This procedure should be used between the Assign and Reset/Rewrite procedures, or immediately after them. In other cases the already buffered data will be lost.

This procedure cannot be used with the address operators.

Example: To expand the standard output text file buffer to 256 characters, the beginning of a program could be written:

```
VAR
  buf :STRING;
BEGIN
  SetTextBuf(output,buf,SizeOf(buf));
  ...
END.
```

Sin

Declaration: **`Sin (x:Real) :Real`**

Description: Calculate the sine of the real type parameter.

Example: `res:=Sin(x);`

SinH

Declaration: **`SinH (x:Real) :Real`**

Description: Calculate the hyperbolic sine of the real type parameter.

Example: `r:=SinH(1.5);`

SizeOf

Declaration: **`SizeOf (t) :Word`**

Description: Return the size that variable t uses, or a variable of type t would use. This function will always return a constant value, and can therefore be used in constant expressions.

No address operators can be used with this function.

Example: see New and SetTextBuf.

Sqr

Declaration: **`Sqr (x:int) :int`**
`Sqr (x:Real) :Real`

Description: Return the square of the integer or real type parameter. The return type will be the same as the parameter type. Type casting of the parameter might be needed if the result can be larger than the parameter type can hold.

For integer types, this function can be used in a constant expression.

This function cannot be used with an address operator.

Example: This program will write the square and square root of ten numbers.

```
VAR
  i :Word;
BEGIN
  FOR i:=1 TO 10 DO
   WriteLn(i:2,’ ^2=’,Sqr(i):3,
           ’, sqrt=’,Sqrt(i):255);
END.
```

Sqrt

Declaration:	`Sqrt (x:Real) :Real`
Description:	Return the square root of the real type parameter.
Example:	see Sqr.

Str

Declaration: `Str (x; VAR s:STRING)`

Description: Convert the integer or real parameter x to a string. If the converted string is too large for the destination string, the string is truncated.

An integer parameter can have an optional length specifier:

```
int:len
```

This specifies the minimum length of the resulting string. To fill the string, spaces are added to the beginning of it. If the length specifier is omitted, the string will be converted to be as packed as possible.

A LongInt expression can be converted to a hexadecimal number, if the expression is preceded by a percent sign. Either a LongInt variable or expression must be used, or the expression must be explicitly type casted to LongInt type. Also here, the length specifies the minimum size, but the number is zero-filled up to a total of eight characters before spaces are added to it.

A real parameter can also have an optional length specifier. In addition, an optional decimal specifier can be used:

```
real:len
real:len:dec
```

If the decimal specifier is omitted, or a negative value is entered, the real number is converted in scientific format. If the length is also omitted, the conversion will be to a 19-character string. A length of 255 will display the number as packed as possible, with suppression of the last decimals of zero value. For numbers with an exponent outside of the range –4 to 10, the packed format will display the number in scientific format.

The decimal specifier will convert the real number as a normal number, with the specified number of decimals. The number will be truncated, not rounded.

This procedure cannot be used with an address operator.

Example: The following program part will read a string. If an empty string is entered, the real number in r will be converted to the string. Last, the string is converted to a real number, checking for any conversion errors.

```
VAR
  r,rt :Real;
  s :STRING;
  p :Byte;
BEGIN
  ...
  REPEAT
    Write('Enter number:');
    ReadLn(s);
    IF s='' THEN Str(r:255,s);
    Val(s,rt,p);
    IF p>0 THEN
     WriteLn('Illegal number')
    ELSE r:=rt;
  UNTIL p=0;
  ...
END.
```

Succ

Declaration: **`Succ (x:Type) :Type`**

Description: Return the next element in the type of variable x. The value will be x+1. The return type is the same as the parameter type.

This function can be used in constant expressions. The address operator cannot be used with Succ.

Example: see Pred.

Swap

Declaration: **`Swap (x:Word) :Word`**

Description: Swap the high and low bytes of the Word type parameter.

This function can be used in constant expressions. The address operator cannot be used.

Example: see Pred.

Tan

Declaration: **`Tan (x:Real) :Real`**

Description: Return the tangent of the real type parameter.

Example: `x:=Tan(Pi/4);`

TanH

Declaration: **`TanH (x:Real) :Real`**

Description: Return the hyperbolic tangent of the real type parameter.

Example: `x:=TanH(rVal);`

Trunc

Declaration: **`Trunc (x:Real) :LongInt`**

Description: Return the real type parameter as a LongInt type, by truncating the fractional part of the real number. No error is generated if the real number is too large.

This function can be used in constant expressions. The address operator cannot be used.

Example: `intVal:=Trunc(Pi);`

UpCase

Declaration: **`UpCase (ch:Char) :Char`**

Description: Return the upper case equivalent of the character ch. The character set bit of the current attribute field (see page 299) specifies which characters are to be converted. For the japanese character set, only a to z are converted. For the ANSI character set, the national characters in the range #224 to #254 are also converted.

Example: If the *Y* key is pressed (either lower or upper case), then erase the file.

```
USES Crt;
VAR
  ch:Char;
  f :FILE;
BEGIN
  ...
  ch:=UpCase(ReadKey);
  IF ch='Y' THEN Erase(f);
  ...
END.
```

Val

Declaration: **`Val (s:STRING; VAR x; VAR pos:Byte)`**

Description: Convert the string s to an integer or real type, as specified by variable x. If an illegal character is found in the string the corresponding string position is set in pos. If all of the string could be converted, pos is set to zero.

The conversion can be made from a hexadecimal number to a LongInt variable, if the numeric variable is preceded by a percent sign.

This procedure cannot be used with the address operators.

Example: see Str.

Write (typed file)

Declaration: **`Write (VAR f:File; VAR v1; ...)`**

Description: Used with a typed file, the Write procedure writes the contents of one or more variables to the file. The type of the variable must be the same as the element type of the file.

The file variable must have been assigned a file specifier, and the file be opened before this procedure can be used.

This procedure cannot be used with an address operator.

Example: see chapter 27.

Write, WriteLn (text file)

Declaration:

```
Write (v1; ...)
Write (VAR t:Text; v1; ...)
```

Description:

This procedure is used to write integer, real, character, string, and boolean expressions to a text file. Either text file t will be used, or the standard output text file. The text file must have been assigned a file specifier, and the file been opened by Rewrite before using this procedure.

All expressions can have an optional length specifier:

```
v1:len
```

This specifies the minimum length to be used for the written text. If the text to output will be larger than this, all of the string is written. If the string is shorter, spaces are added to the beginning of the string. If no length is specified, the string is written with the minimum number of characters possible.

An integer expression is decoded in the same way as in the Str procedure. Also here, hexadecimal LongInt expression can be written by preceding the expression by a percent sign.

Real expressions can have both an optional length, and a decimal specifier.

```
real:len
real:len:dec
```

The length and decimals are decoded in the same way as in procedure Str.

A boolean expression will write either of the texts **FALSE** or **TRUE**, depending on the value of the expression. If a length longer than this is specified, spaces are added in front of the text.

This procedure cannot be used with the address operators.

Example: see chapter 27.

Chapter 32

Crt Unit

The Crt unit includes some special variables and subroutines when reading from the keyboard and writing to the display. A special text file driver is also defined, to speed writing to the display. When the Crt unit is used, the standard output file is automatically set to this fast output driver, with any redirection disconnected.

Crt Variables

The following variables are defined in the Crt unit.

dispStatus

Declaration: `dispStatus :Byte`

Description: This flag byte specifies some flags used when reading and writing to the display. To use another setting of this byte, add the wanted field values together and assign the summed value to the variable. The default value at the beginning of a program is zero.

The first field controls the input line function when a character key is pressed as the first key. The value 0 is used to clear the input line for a first character key. Value 1 will never clear the input line, regardless of whether a cursor or character key is pressed as the first key.

The next field restricts display scrolling. A value of 0 is used for normal scrolling when the cursor moves past the right margin of the display or window. The value 2 specifies that no scrolling or line change should be made when the cursor moves past the right margin. This flag setting can be used to be able to write to the bottom right character position without scrolling the screen.

The last field controls the writing of real numbers. The value 0 is used to write a decimal point, and the value 4 is used to write a decimal comma. In a program, the Read and Val procedures always allow both a decimal point and comma as the decimal separator.

Example: This example will set the decimal comma flag, and then write a real value.

```
USES Crt;
VAR r :Real
BEGIN
  r:=12.345;
  dispStatus:=4;
  Write(r:255);
END.
```

keyStatus

Declaration: `keyStatus :Byte`

Description: This flag byte specifies the flags used by the keyboard decoding routine. To change a field, mask this bit by the AND operator, and add the wanted value. The fields not specified here are reserved for internal use. Any changes of these reserved fields might disable some other function of the computer.

The first field controls the reading of the programmable keys. The mask for this field is 254. A value of 0 will enter the programmed key sequence in the key buffer when that key is pressed. The value 1 is used to insert the key code instead, as a two-key sequence of $00,$D0 through $00,$E3 (depending on the key pressed; see page 292 for details).

The next field specifies the key beeper. The mask is 253, and the value 0 will set the key beeper off. Value 2 sets the beeper on.

The last field that can be changed is the repeat flag. The mask value for this field is 251. When set to value 0, no repeat will be made on the last key. By setting the value to 4, the last key will be repeated if kept pressed. This is the default value when a cursor key is pressed.

This field is automatically set or cleared when a new key is pressed.

textAttr

Declaration: `textAttr :Byte`

Description: The current display attribute is set in this byte. This is used by the subsequent Write procedures when writing to the display. To change the attribute, add the new attribute fields together, and assign it to the variable.

The first field specifies underlining. A value of 0 sets underlining off, and value 1 sets it on.

Next field is used for dimmed characters. Value 0 sets normal characters, and value 2 sets dimmed characters.

Reverse characters are controlled by the following field. For normal characters, the value 0 is used. Value 4 will set reversed characters.

The last field specifies which character set to use. The value 0 is used for the ANSI/ISO character set, and value 128 for the japanese character set.

Example: This program part will write the first text in normal way, and the second one using underline.

```
...
Write('Normal, ');
textAttr:=1;
WriteLn('Underline');
textAttr:=0;
...
```

Crt Procedures and Functions

The following procedures and functions are declared in the Crt unit.

AssignCrt

Declaration: `AssignCrt (VAR t:Text)`

Description: Set text file variable t to the fast display writing device routines. This is automatically done for the standard output text file.

Before the file can be used to write data to, it must be opened by the Rewrite procedure. This has automatically been done for the output text file when the Crt unit is imported.

Example: Open a special file to use the fast display routines in the Crt unit.

```
USES Crt;
VAR
  ct :Text;
BEGIN
  AssignCrt(ct);
  Rewrite(ct);
  ...
END.
```

ClrEol

Declaration: `ClrEol`

Description: Clear the current line where the cursor is located. The character positions at the cursor, up to the end of the line in the window, are cleared.

Example: The following program will read five numbers to an array. All numbers are read at the same place in the display. ClrEol is then used to remove the previously entered string.

```
PROGRAM ClrDemo;
USES Crt;
VAR
  a :ARRAY [1..5] OF Word;
  p :Word;
BEGIN
  ClrScr;
  WriteLn('Enter number:');
  FOR p:=1 TO 5 DO BEGIN
    GotoXY(1,2);
    ClrEol;
    ReadLn(a[p]);
  END;
  ...
END.
```

ClrScr

Declaration: `ClrScr`

Description: Clear the current window.

Example: see ClrEol.

Cursor

Declaration: **`Cursor (on:Boolean)`**

Description: Set the flashing cursor on or off. A parameter value of True will set the cursor on.

The cursor must be turned off before writing text to the display. This is not done automatically. If the cursor is shown while writing to the display, the operation of the LCD display might be interrupted or disabled.

The cursor is automatically switched off at the end of the Read procedure and ReadKey function.

Example: The following function will display the cursor and wait for a key, while incrementing a variable.

```
Uses Crt;
VAR
  counter :Word;
FUNCTION WaitKey :Char;
  BEGIN
    Cursor(TRUE);
    WHILE NOT KeyPressed DO
     Inc(counter);
    WaitKey:=ReadKey;
  END;
```

DelLine

Declaration: **`DelLine`**

Description: Delete the current cursor line, and move the following lines in the window one line up. The last line in the window is cleared.

Example: Move the second line in the window to the top line, and clear the second line. Other lines are to remain in their lines. This is solved by using DelLine to delete the top line, and InsLine to insert a new second line (thereby restoring the third and fourth lines in the window).

```
USES Crt;
BEGIN
   ...
   GotoXY(1,1);
   DelLine;
   GotoXY(1,2);
   InsLine;
   ...
END.
```

Display

Declaration:	**`Display`**
Description:	Copy the current graphic buffer data to the LCD display. Only those parts that has been marked as changed are copied. This can manually be set by the UpdateRegion procedure.
Example:	see chapter 30.

GotoXY

Declaration:	**`GotoXY (x,y:Byte)`**
Description:	Position the cursor in the current window. The upper left corner of the window has coordinates (1,1). If either parameter is outside of the window, the cursor position is not changed.
Example:	see chapter 29.

InsLine

Declaration:	**`InsLine`**
Description:	Insert a line at the cursor line, and move the cursor line and the following lines one line down. The bottom window line is deleted, and the inserted line cleared.
Example:	see DelLine.

KeyPressed

Declaration:	**`KeyPressed :Boolean`**
Description:	Return the current keyboard buffer status. If a key is set in the keyboard buffer, True is returned.

Example: see Cursor.

Line

Declaration: **`Line (x1,y1, x2,y2:Integer; col:Byte)`**

Description: Draw a line in the display, from coordinate (x1,y1) to (x2,y2). The line is clipped if any part is outside of the display. The top left pixel of the display has coordinate value (0,0).

The color of the line is entered to the col parameter. A value of 0 will put each pixel off. The value 1 will set the pixels on, and value 2 will invert each pixel in the line (on to off and vice versa).

This procedure only updates the graphic display buffer. To actually display the graphics data, the Display procedure must be used. This allows a number of graphic elements to be drawn, and then displayed on the LCD all at the same time.

Example: see chapter 30.

NextKey

Declaration: **`NextKey :Word`**

Description: Read the next key from the keyboard buffer, without removing it from the buffer. If no key is found, the value zero is returned. Otherwise, the ordinal value for that character is returned. A double-code key will set the low byte to zero, and the high byte to the actual code. These codes can be read by using the Lo and Hi functions.

Example: Only if the next key is a digit, that key should be read and decoded.

```
USES Crt;
VAR
  key :Word;
  inData :Integer;
BEGIN
  ...
  key:=NextKey;
  IF (key>=Ord('0')) AND (key<=Ord('9'))
   THEN BEGIN
    ReadLn(inData);
    ...
  END;
  ...
END.
```

Plot

Declaration:	`Plot (x,y:Integer; col:Byte)`
Description:	Plot a pixel in the display at coordinate (x1,y1). Nothing is set if the coordinate is outside of the display. The top left pixel of the display has coordinate value (0,0). The color of the pixel is entered to the col parameter. A value of 0 will set the pixel off. The value 1 will set the pixel on, and value 2 will invert the pixel value (on to off and vice versa). This procedure only updates the graphic display buffer. To actually display the graphics data, the Display procedure must be used. This allows a number of graphic elements to be drawn, and then displayed on the LCD all at the same time.
Example:	see chapter 30.

Point

Declaration:	`Point (x,y:Integer) :Byte`
Description:	Get the value of the pixel at coordinate (x,y). If this pixel is outside of the display, the value 0 is returned. For pixels within the display, a set pixel will return value 1 and a pixel that is off will return value 0. The upper left pixel of the display has coordinate (0,0).
Example:	see chapter 30.

ReadKey

Declaration:	`ReadKey :Char`
Description:	Get the next character from the keyboard buffer. For a double-code key, two readings are needed. The first one returns value 0, and the second the actual code of that key.
Example:	see Cursor and NextKey.

Sound

Declaration: **`Sound (freq,dur:Word)`**

Description: Emit a tone in the internal speaker. The tone will have the approximate frequency freq (in Hz), and of length dur (in ms).

Not all frequencies can be generated, due to the timing of the computer. High frequencies (over 4000 Hz), and very short tones, will not emit any tone at all.

Example: The following program makes a special sound in the speaker.

```
PROGRAM SoundDemo;
USES Crt;
VAR
  w :Word;
BEGIN
  w:=400;
  WHILE (w<3000) DO BEGIN
    Sound(w,50);
    Inc(w,w DIV 20);
  END;
END.
```

TextDisplay

Declaration: **`TextDisplay`**

Description: Convert all of the character display buffer to graphics data (in the graphic buffer), and then display that data on the LCD. The previous contents of the graphics buffer will be deleted by this procedure.

Example: see appendix D.

TimerOut

Declaration: **`TimerOut (timer:Byte) :Boolean`**

Description: Check if the specified timer has timed out. The timer number can be 0, 1, or 2. The function will return True if the timer has counted down to zero.

Example: Start timer 0 for a delay of one second, and wait until it has timed out. Meanwhile, increment a variable.

```
USES Crt;
VAR
  counter:Word;
BEGIN
  ...
  counter:=0;
  TimerStart(0,10);
  WHILE NOT TimerOut(0) DO
   INC(counter);
  ...
END.
```

TimerStart

Declaration: **`TimerStart (timer:Byte; count:Word)`**

Description: Start the timer to count from the start count down to zero. The countdown is made at approximately 10 Hz. Use function TimerOut to check if the timer has counted down to zero.

The timer number can be 0, 1 or 2.

Example: see TimerOut

UpdateRegion

Declaration: **`UpdateRegion (x1,y1, x2,y2:Integer)`**

Description: Set the update region for changes in the graphics buffer. This is needed if the graphic buffer is changed directly, without using procedures Plot or Line. The procedure Display is then used to copy the contents of the graphic buffer region to the LCD.

The upper left pixel of the display has coordinate (0,0).

Example: see chapter 30.

WhereX

Declaration: **`WhereX :Byte`**

Description: Return the current cursor position within the window. The first character position in the window has value 1.

Example: see chapter 29.

WhereY

Declaration:	`WhereY :Byte`
Description:	Return the current cursor line within the window. The first line in the window is line 1.
Example:	see chapter 29.

WindMaxX

Declaration:	`WindMaxX :Byte`
Description:	Return the last character position of the current window. This is a value between 1 and 32.
Example:	see chapter 29.

WindMaxY

Declaration:	`WindMaxY :Byte`
Description:	Return the last line in the current window. This is a value between 1 and 4.
Example:	see chapter 29.

WindMinX

Declaration:	`WindMinX :Byte`
Description:	Return the first character position of the current window, as a value between 1 and 32.
Example:	see chapter 29.

WindMinY

Declaration:	`WindMinY :Byte`
Description:	Return the first line of the current window, as a value between 1 and 4.
Example:	see chapter 29.

Window

Declaration:	`Window (x1,y1, x2,y2:Byte)`
Description:	Set a new text window, at the display coordinates (x1,y1) to (x2,y2). The upper left corner of the display has coordinate (1,1). If the specified coordinates are outside of the window, the current window is not changed. Subsequent reading and writing of text to the display will only be made within the window.
Example:	see chapter 29.

Chapter 33

Compiler Options

A *compiler option* is entered in a comment, with a dollar sign as the very first character in the comment. The identifier immediately following the dollar sign specifies the actual compiler option. There must be no space between the comment start symbol and the dollar sign, or the dollar sign and the first identifier. If the identifier is followed by another identifier, they must be separated by at least one space, otherwise the two identifiers will be read as just one identifier.

Examples:

```
{$R+}
(*$I Include.Fil   Normal comment *)
{$S-,T-,F-}
```

There are three different types of compiler options: *Compiler flags*, *Conditional compilation*, and *Compiler directives*.

Compiler flags

The compiler flags are used to control the code being generated. If more than one flag need to be changed at one instance, they can be grouped in one comment, with commas between each flag and no spaces (see the example above). The flag is set by using a plus sign (+), and reset by a minus sign (–).

A flag can be restored to the value it had immediately before the last setting or resetting of that flag, by using an equal sign instead of a plus or minus sign.

Example:

The following code will remove division by zero error testing during execution of the expression, and restore the state of the flag immediately thereafter:

```
{$F-}
var1 := (value + 1) DIV (value - 1);
{$F=}
```

The compiler flags are the following (the default state is shown first):

C+ C–

Set subroutine copy of value-declared structured types on or off. When this flag is reset, no copy of the parameters is done, and the result within the subroutine is as if the parameter is var-declared. The parameter can still be called with a constant value (e.g. a string constant). The called parameter must have been declared in bank 1 (a short pointer).

Normally this option is only used when maximum speed and minimum stack size is wanted, and where the parameter variable is not changed, but where the procedure or function will be called with a constant value.

The flag should be restored at the end of the subroutine where it was used, so that further subroutines behave in a standard way.

D+ D–

Set debug information generation on or off. When the flag is reset, no information is generated about variables, procedures, functions, or line numbers. The debugger cannot be used for variables or code compiled with the flag reset.

With this flag set, the code file size increases by about 50%. Normally, this flag is therefore only set during program development, and reset when compiling the final version of the program.

In large programs, the code file size can be reduced by selectively setting this flag on only at those parts in the program needing debugging. In addition, the main part of the program should have the debug flag set, so that the debugger can be used in the normal way.

F+ F–

Generate floating point overflow and division by zero error tests where appropriate when this flag is set. If the flag is reset, the programmer has to make the necessary error tests in order to verify the validity of the result. When the flag is set, execution terminates if a run-time error is detected.

I+ I–

Set file error checking on or off. When this flag is set, a file operation error terminates execution immediately. With the flag reset, the programmer must call the IOResult function to get the result of the last file operation.

This compiler option is also used for file inclusion. This is explained at compiler directives below.

R– R+

This flag handles the range error checking. When set, execution terminates when assignment of too small or large values are detected. No errors will be generated when this flag is reset.

S+ S–

Stack overflow testing is enabled by this flag. When the flag is set, the program terminates when the system or user stack overflows. This can be the result of too deep recursion and/or too much usage of the heap.

When the flag is reset, no errors are generated. This can then result in the stack growing too large, and destroying other areas in the computer, possibly needing a complete reset of the computer.

T+ T–

Dereferenced pointer error testing is handled by this flag. When the flag is set, an error is generated when a NIL pointer is dereferenced. No error is generated if the flag is reset.

V+ V–

The last compiler flag controls type checking at variable-declared string parameters. The flag is set when strict type checking is wanted (i.e. the called string parameters must have identical lengths as the declared ones). With the flag reset, a string parameter of any length can be used. In this case, it is possible that the subroutine might be writing at a position outside of the string, if the length of the called parameter string is too small.

Conditional compilation

The conditional compilation makes it easy to skip parts of the code at compilation time. Some code may be skipped when e.g. the debug flag is reset, and other code may only be included when the code is compiled on a certain machine, or even a certain compiler version.

For the conditional compilation, *conditional identifiers* can be defined and/or undefined. These identifiers are completely separated from the identifiers in the normal Pascal program. As such, an identifier declared as a conditional identifier will not be available in the actual Pascal program (a compilation error will be generated), and vice versa. But it is fully possible to declare an identifier of the same name as both a conditional identifier and a Pascal program identifier (in which case they still will be located and interpreted in different ways).

There are three predefined conditional identifiers. These are: DLPASCAL (the type of Pascal compiler), V12 (the current version number), and PB2000 (the computer type).

Examples:

```
{$DEFINE NewId}
(*$UNDEF NotLongerUsed *)
{$IFDEF NewId}
  WriteLn ('This will be written');
{$ELSE}
  WriteLn ('This will not even be compiled'
{$ENDIF}
{$IFOPT D+}
  WriteLn ('Only written if debug flag is set');
{$ENDIF}
```

DEFINE identifier

This compiler option defines a new conditional identifier, to be used for conditional compilation further on. If the identifier has already been declared, the option will just be skipped.

The identifier to be defined must be separated from the DEFINE keyword by at least one space.

UNDEF identifier

A conditional identifier already defined can be undefined by this option. Any reference to that identifier in later compiler options will not find that identifier. If the identifier has not been declared, this compiler option is just skipped.

The identifier must be separated from the UNDEF keyword by at least one space.

IFDEF identifier

This option is used when the Pascal source code following this option is to be compiled only if the conditional identifier is defined. If the identifier is not defined, the source code will be skipped until either of the ELSE or ENDIF keyword is found.

IFNDEF identifier

Analogous to the IFDEF option, this option is used then the following source code is to be compiled only if the conditional identifier is not defined.

IFOPT flag+
IFOPT flag–

Only if the compile flag is set (flag+) or reset (flag–), the source code following this option is compiled. Otherwise, the source code is skipped until an ELSE or ENDIF keyword is found.

ELSE

The ELSE option is used to invert the skip state; i.e. if the source code was being skipped as the result of the latest conditional compilation test, the source code between the ELSE and the ENDIF will be compiled, and vice versa.

If the latest conditional test itself was skipped due to an earlier conditional test, the ELSE keyword has no function.

ENDIF

This option ends the latest conditional test. If the code was being skipped as the result of this test, compilation continues from this point in the source code.

Compiler directives

The last two compiler options are used to include another file to be compiled, and to set the run-time heap and stack area sizes.

I includeFile

Another file can be included for compilation by this option. The file is specified after the option (with at least a space inbetween). The included file can itself include another file, up to a nested level of three files in addition to the main file.

For procedures and functions declared in an include file, no line number information is generated. This means that no break points can be set in those subroutines.

The same option identifier is used to control file error checking; see compiler flags for more detail about this.

M heapSize
M heapSize,stackSize

This option is used to force the heap and/or stack area sizes to a certain size when the program is run. The size is specified in bytes. A value of 0 is used when the size is not to be changed (the default is no change of neither the heap size nor the stack size: $M 0,0). There must be at least one space between the M and the heap size.

The memory option only works in a program. It has no function when used in a unit.

Chapter 34

Inside PB-2000 DL-Pascal

This chapter documents the formats used to store variables and parameters. Also the function return variable is described.

In the PB-2000, there are four *banks* or *segments*, numbered 0 through 3. Bank 0 is the ROM card bank, and bank 1 the normal RAM memory. Also, bank 3 at address $0000 through $7FFF is used by the ROM card. Bank 2 and the remainder of bank 3 are not used. Near pointers always use bank 1. By using far pointers or absolute variables, the other banks can be accessed.

Data Formats

Variables are stored in the memory end-to-end to each other. Global variables are allocated space on the heap when the program is started. Local variables are allocated space when the subroutine is called, on the user stack (using the **US** stack register).

Ordinal Types

Simple variables occupy little space (size is in bytes):

Byte	1
ShortInt	1
Word	2
Integer	2
LongInt	4
Enumerated	1 or 2
Char	1
Boolean	1

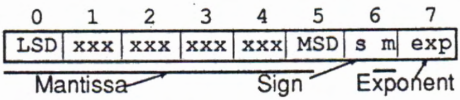
Enumerated types occupy one byte, if the total number of elements is less than or equal to 256.

Variables declared as a subrange of a type occupy the same size as the base type of the subrange.

Variables are stored with the least significant byte at the lowest address.

Real Types

The Real type occupy eight bytes (or 16 half-bytes) as a BCD number, as can be seen in the figure to the right. The exponent is stored as a binary number, biased by 128. An exponent of zero signals that the whole number has the value zero. The sign half-byte has value 0 for a positive, and 1 for a negative sign. In addition, arithmetic overflow is stored in this position. If an overflow has occurred, this value is 2. An argument error sets the value to 3, and division by zero sets it to 6.



Pointer Types

A near pointer is stored as a Word, implying a bank value of 1. A far pointer is stored as two Words, with the bank number as the second Word.

Procedure and Function Types

Subroutine types are stored as far pointers, using two Words with the bank number in the second Word.

String Types

A string is stored as a string length byte, followed by the actual string holder. The total size of a string type is one more than the declared maximum string length, to make space for the string length byte.

Array Types

An array is stored as a sequence of the base type of the array. For a multi-dimensional array, the rightmost index increases first.

Record Types

Variables of record type are stored in consecutive order according to the declaration. The variant lists of a record will always start at the same address within the record.

Set Types

Sets are stored as a sequence of bits. The first element bit is stored at the lowest bit (bit number 0). The number of bits used are specified by the base type of the set, regardless of any subrange type. The number of bytes used to store these bits can be calculated as:

```
(Max(baseType)+7) DIV 8
```

For a set of an integer or character type, the set variable always occupies 32 bytes.

File Types

A file type variable occupies 32 bytes of memory. It can be thought of having the following record definition:

```
FileType = RECORD
             handle  :Byte;
             mode    :Byte;
             opCode  :Byte;
             device  :Byte;
             fileType:Byte;
             fileName:ARRAY [0..10] OF Char;
             res     :ARRAY [0..15] OF Byte;
           END;
```

The field mode is set to zero if the file is closed, and 1 for an open file.

A text type variable uses 96 bytes. The record definition for this type is:

```
TextType = RECORD
             handle  :Byte;
             mode    :Byte;
             opCode  :Byte;
             device  :Byte;
             fileType:Byte;
             fileName:ARRAY [0..10] OF Char;
             bufSize :Word;
             bufRead :Word;
             bufWrite:Word;
             bufAddr :Word;
             textCall:FUNCTION(VAR t:TextType):Byte;
             res     :ARRAY [0..3] OF Byte;
             buffer  :ARRAY [0..63] OF Char;
           END;
```

The field mode is zero when the text file is closed. A value of 1 specifies that the file has been opened by Reset, 2 if opened by Rewrite, and 3 if opened by Append.

The other fields are described on page 200, forward.

Parameter Formats

Parameters to a subroutine are always accessed with the first parameter at the lowest address. Access to the parameters are done using a base pointer in the IX register. If the subroutine is declared at the global level, the first parameter is at IX+0. For lower levels of subroutines, the first parameters is at IX+2. In this case, the dynamic link (the base pointer of the calling subroutine) is stored at address IX+0.

The subroutine's start and end code reserves enough space to hold all local variables, temporary space, and return variable (for functions). The return variable is accessed differently depending on the return type:

Byte	IX–3
ShortInt	IX–3
Word	IX–4
Integer	IX–4
LongInt	IX–6
Real	IX–10
Char	IX–3
Boolean	IX–3
Enumerated	IX–3 or IX–4
Pointer	IX–4
LongPtr	IX–6
String	IX–4

When calling a function that will return a string type, the caller must reserve space on the stack for the return string, and send the address to that space to the subroutine in registers $4-5. This address is then stored at the return variable area in the subroutine.

Local variables are stored directly following the return variable. For procedures, the first local variable is stored at the place of the return variable, with the other local variables directly following it.

Variable-declared parameters always take 2 bytes. This is the address to the actual variable contents. Therefore only variables declared in bank 1 can be used at variable-declared parameters.

Value-declared parameters normally take the same memory space as the corresponding variables. For parameters of set, record, array, and string types, the address to the parameter is pushed as the parameter, instead of the parameter itself. This address is stored as a far pointer, using 2 Words with the offset first. The subroutine must make a local copy of such parameters. This is automatically done by the start code of the subroutine, but can be disabled by the {$C–} compiler option.

Register Saving

In a subroutine or INLINE code, registers $00-$23, IY, and IZ can be freely used. IX and $28-29 both are used as the base pointer to parameters and local variables, and both must be preserved.

Chapter 35

Comparing to Other Pascals

The Pascal language in the DL-Pascal compiler is very close to standard Pascal. However, due to the restricted memory available on the ROM card, there are some small changes from it.

DL-Pascal is also quite close to the de-facto standard Pascal compilers on the PC market: Turbo Pascal 5.5 and QuickPascal 1.0. Also here, because of the restricted memory space, not every feature has been possible to implement.

Standard Pascal

The following changes from standard Pascal have been made in DL-Pascal:

- String constants are only compatible with the string type, not with a PACKED ARRAY OF Char.
- In a variant record, the colon separating the tag field identifier and the tag type must always be entered, even if only the tag type is specified.
- Procedure and function types are allowed, and not only as parameters. The syntax is different, though. The subroutine type must be declared separately, and that type is then used as the parameter type.
- Labels can only be identifiers, not the numbers the standard specifies.
- The GOTO statement can only jump within a subroutine, not outside of the current subroutine.
- The identifier used in a FOR statement must be a simple variable declared in the current subroutine, or a simple global variable.
- Variable tag fields in the New and Dispose procedures are not allowed.
- The reserved word PACKED has no function, and therefore the procedures Pack and Unpack are not implemented.
- The file type Text is a special type, not declared as a FILE OF Char.

- The file buffer is not implemented in the way the standard specifies, using the file variable as a pointer. The file functions Get and Put are also not implemented. Instead, the Read and Write procedures must be used.

In addition to these changes, there are a number of enhancements made to the standard. Some of the more important ones are:

- Separate compilation of units.
- Integers of different sizes.
- Constant expressions can be used instead of just constant values.
- Initialized variables can be declared.
- Parameters of a subroutine can be an open array or untyped.
- Declarations can be made in any order, not just the strict order defined by the standard.

Turbo Pascal and QuickPascal

Most of the features found in Turbo Pascal and QuickPascal are implemented in DL-Pascal. However, there are some differences from those compilers:

- Objects are not implemented.
- Typed constants are not implemented. Instead, initialized variables are used in the same way.
- A set type occupies the size according to the common base type, regardless of any subrange used.
- Only one real type is used, and this is implemented using BCD arithmetic.
- Control characters in strings can be included using the pound sign (#), but not using a caret (^).
- String and set operations cannot be used in constant expressions.
- A function call returning a pointer cannot be used on the left side of an assignment.
- The use of double address operators for procedure variables is not implemented. Instead, a special address operator is implemented to always return the address of a variable.
- Procedures and functions cannot be declared as INTERRUPT, EXTERNAL, or INLINE.
- Values in INLINE statement are separated by commas instead of the division sign.

- The interface and implementation parts of a unit are split into two files. The actual syntax for each unit heading is somewhat different, but the declarations and implementations are declared in the same way.
- Some compiler options are not implemented or are different (A, B, E, F, L, N, O), and three options are added (C, F, T).
- The System and Crt units omits some variables and some special functions. Further, the Printer, Dos, Graph, Overlay, Turbo3, and Graph3 units are not implemented. Instead, some graphic subroutines are implemented in the Crt unit.

There are even some additions made to the DL-Pascal compiler:

- National characters can be used in identifiers.
- Open arrays can be used as parameters, to be able to use arrays of any size but still have type checking.
- Both near and far pointers can be used.
- The System unit has some procedures and functions added, such as mathematical functions. The Pos functions is enhanced to take an optional third parameter, specifying the start position of the search.

Part 4

Appendix

Appendix A

Batteries

There are two different types of batteries in the PB-2000. The *main battery* is used for the main power supply of the computer. The *backup battery* is only used to supply power to the RAM memory, in case the main battery cannot deliver enough power. The backup battery is one CR1220 lithium cell, and should be replaced every other year. The main battery has a life time of around 30-40 hours.

Checking the Power

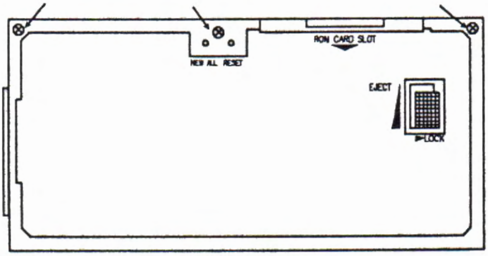
If you cannot get any power to the PB-2000, check that the ROM card lock switch on the back side of the computer is in LOCK position. Also press the ON BRK key, as this is used to switch power on after an automatic power off. If you still cannot get power to the computer, it might be needed to press the ○ RESET button.

The main battery consists of three CR2032 lithium cells, and should be replaced as soon as the display gets too weak to be read comfortably. You should first check the contrast setting by pressing the S CONTRAST / keys and the arrow keys ⇧ and ⇩. If this does not help, the main battery cells have to be replaced. Because of the internal structure of lithium batteries, all three main battery cells should be replaced at the same time.

When changing batteries, only one battery type should be replaced at one time. This will assure that the memory of the PB-2000 will not be changed. If both batteries are replaced at the same time, or if the other battery has been discharged too low to supply any power at all, the memory contents will be deleted. Since the batteries contain material poisonous to the environment, they should not be tossed in the normal garbage. Also, the batteries must not be charged, short circuited, disassembled, burned, or swallowed. Keep them out of reach of small children.

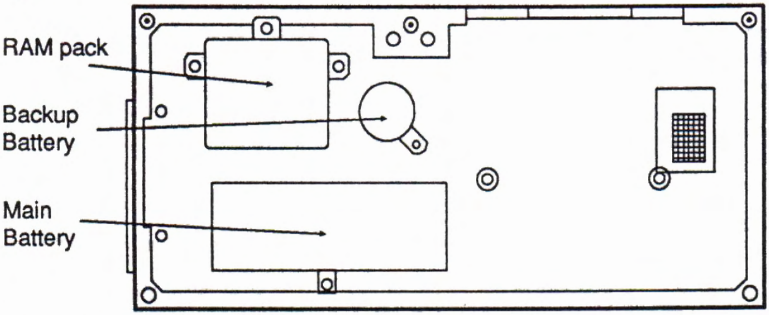
Changing Batteries

To change the batteries, the back cover of the computer has to be removed. Make sure the power of the PB-2000 has been switched off, turn the computer upside down,

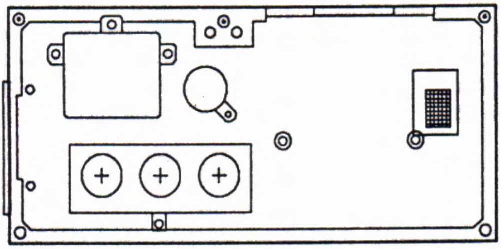


and unscrew the three screws holding the back cover in place. Then remove the back cover.

There are three compartments inside the PB-2000. One is for the optional RAM packet, and the other two is for the batteries. Locate the main or backup battery compartment, depending on which battery you want to replace.



Remove the cover to the battery compartment, by unscrewing the screw holding the cover in place. Lift and flip the computer over, and the old batteries should fall out. Then place the computer upside down again.



Insert the new batteries, after wiping them off with a soft cloth. You should not handle the batteries directly with your hands, as the grease can shorten the life of the batteries, and also make bad contact. Use a cloth instead. Make sure the positive side of the battery is facing upwards (a positive sign should be visible on all batteries).

Replace the battery compartment cover, and screw the screw in place.

Last, replace the back cover of the PB-2000, making sure the small tongues of the back cover fits correctly. Screw all three screws in place. Make sure the batteries work, by switching the power of the computer on.

Using an AC adapter

To keep battery consumption low, an *AC adapter* can be connected to the computer when a mains connector is available. The AC adapter should be the **AD-4175** adapter available from CASIO.

To connect the AC adapter, the power of the PB-2000 should be switched off. Insert the AC adapter to the mains power outlet, and the cable from it into the adapter connector of the PB-2000 (located on the left side of the computer).

Appendix B

ROM Cards

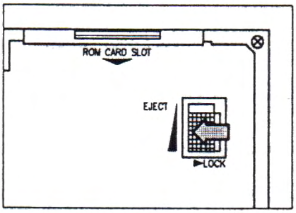
Different programming languages and programs can be used in the PB-2000, by changing *ROM cards*. Currently, the interpreting languages BASIC and PROLOG are available, as well as this compiling Pascal ROM card.

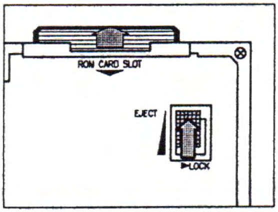
When a new ROM card has been inserted, all non-sequential files are removed from the RAM memory. To be on the safe side, all important files should be saved before switching ROM card.

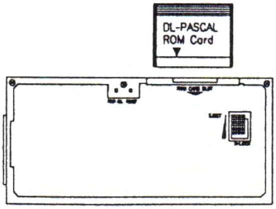
Because of an oversight in the BASIC ROM card, all code and unit files (extensions **EXE**, **DLE**, and **UNI**) should be manually deleted before switching from the Pascal to the BASIC card.

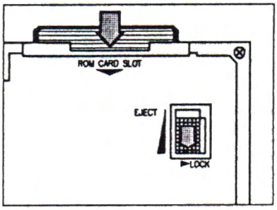
Replacing the ROM card

To replace the ROM card, first switch the power of the PB-2000 off. On the back of the computer, slide the eject switch first to the left, and then up. This will release the old ROM card (or the dummy card if no ROM card has been inserted). Remove the old ROM card from the ROM card slot.





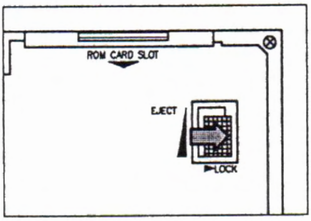




Insert the new ROM card in the ROM card slot, and press the card gently all the way down in the slot. The eject switch will automatically be pushed down. The ROM card will only fit with the small arrow of the card facing the back of the PB-2000.

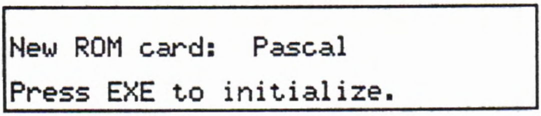
Last, slide the eject switch to the right, as shown by the LOCK arrow.

If no ROM card is set in the computer, the dummy card should be inserted instead to protect the internal connector of the PB-2000. Also, the eject switch must be in LOCK position for the computer to operate.



Starting the ROM card

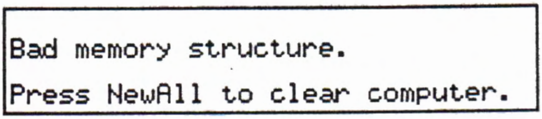
After inserting the ROM card and switching power on, a message is displayed.



```
New ROM card:  Pascal
Press EXE to initialize.
```

After the EXE key is pressed, all non-sequential files will be deleted. If you have some files not yet saved, you can still replace the ROM card to the old one, and save those files.

If the internal memory has been destroyed, or if the system RAM area is too small, the following message is displayed.



```
Bad memory structure.
Press NewAll to clear computer.
```

In this case, the **O NEW ALL** button must be pressed, and all files will be deleted. If just the system RAM area is too small, you can replace the ROM card to the old one, and change the memory allocated to the file area to a smaller value. If the memory contents has been destroyed, however, all you can do is press the **O NEW ALL** button.

Appendix C

Key Codes

The keys on the keyboard return a key code. This is normally a single code, e.g. a letter. In the tables on the next page, these keys are shown with the value of the key in hexadecimal and decimal, followed by the actual character it produces.

Any single key code can be produced by pressing the ANS shift key, followed by a three-digit decimal value. The entered value is then placed in the keyboard buffer.

Some keys return a double-key code (also called *extended* key code). When such a code is read in a program, first a character of value #00 (the *null character*) will be read, followed by the actual key code. Extended key codes are shown in the tables below with the null character followed by the actual key code in hexadecimal and decimal.

The tables give the key codes returned when the key is pressed without any shift code, in combination with the S key, and in combination with the ANS key. Displayed in the tables are also the keys used on a PC, corresponding to the key codes returned from the keyboard in the PB-2000.

Normal Keys

The normal keys are the letters, digits, and symbol keys.

To make umlauted characters, some key combinations produce *dead* keys. They are marked in the table with an asterisk. To make an umlauted character, first press the dead key combination, followed directly by the character to umlaut. Either the normal or the shifted key used for the umlaut has a character similar to the umlaut character, which makes it easier to locate the umlaut keys.

Example:

To make the letter Ä, press ANS *(A*

Normal keys

	Normal		S		ANS		PC
TAB	$09, 9		00 $0F, 15		$1B, 27		Tab, Esc
’	$27, 39	’	$21, 33	!	$B4, 180	’ *	
(	$28, 40	(	$22, 34	"	$A8, 168	" *	
)	$29, 41	)	$23, 35	#	$A3, 163	£	
[	$5B, 91	[	$24, 36	$	$A2, 162	¢	
]	$5D, 93	]	$25, 37	%	$B0, 176	•	
|	$7C, 124	|	$26, 38	&	$A6, 166	|	
q	$51, 81	q	$3F, 63	?	$BF, 191	¿	
w	$57, 87	w	$40, 64	@	$A1, 161	¡	
e	$45, 69	e	$5C, 92	\	$B6, 182	¶	
r	$52, 82	r	$5F, 95	_	$AF, 175		
t	$54, 84	t	$60, 96	‘	$60, 96	‘ *	
y	$59, 89	y	$7B, 123	{	$A5, 165	¥	
u	$55, 85	u	$7D, 125	}	$B5, 181	µ	
i	$49, 73	i	$7E, 126	~	$7E, 126	~ *	
o	$4F, 79	o	$3C, 60	<	$AB, 171	«	
p	$50, 80	p	$3E, 62	>	$BB, 187	»	
=	$3D, 61	=	$5E, 94	^	$5E, 94	^ *	
a	$41, 65	a	00 $D0, 208		$C6, 198	æ	
s	$52, 82	s	00 $D1, 209		$DF, 223	ß	
d	$44, 68	d	00 $D2, 210		$F0, 240	d	
f	$46, 70	f	00 $D3, 211		$DE, 222	p	
g	$47, 71	g	00 $D4, 212		$D8, 216	ø	
h	$48, 72	h	00 $D5, 213		$A9, 169	©	
j	$4A, 74	j	00 $D6, 214		$AE, 174	®	
k	$4B, 75	k	00 $D7, 215		$AA, 170	ª	
l	$4C, 76	l	00 $D8, 216		$BA, 186	º	
;	$3B, 59	;	00 $D9, 217		$B2, 178	²	
:	$3A, 58	:	00 $DA, 218		$B3, 179	³	
z	$5A, 90	z	00 $DB, 219		$D7, 215	œ	
x	$58, 88	x	00 $DC, 220		$C5, 199	å	
c	$43, 67	c	00 $DD, 221		$C7, 201	ç	
v	$56, 86	v	00 $DE, 222		$A7, 167	§	
b	$42, 66	b	00 $DF, 223		$BC, 188	¼	
n	$4E, 78	n	00 $E0, 224		$BD, 189	½	
m	$4D, 77	m	00 $E1, 225		$BE, 190	¾	
,	$2C, 44	,	00 $E2, 226		$B9, 185	¹	
SPC	$20, 32		00 $E3, 227		$A0, 160		

	Normal		S		ANS		*PC*
/	$2F, 47	/	contrast		$B8, 184	,	
*	$2A, 42	*	$2A, 42	*	$A4, 164		
–	$2D, 45	–	$2D, 45	–	$AC, 172	¬	
+	$2B, 43	+	$2B, 43	+	$B1, 177	±	
0	$30, 48	0	$30, 48	0	code		
.	$2E, 46	.	$2E, 46	.	$B7, 183	•	
1	$31, 49	1	$31, 49	1	code		
2	$32, 50	2	$32, 50	2	code		
3	$33, 51	3	$33, 51	3	code		
4	$34, 52	4	$34, 52	4	code		
5	$35, 53	5	$35, 53	5	code		
6	$36, 54	6	$36, 54	6	code		
7	$37, 55	7	$37, 55	7	code		
8	$38, 56	8	$38, 56	8	code		
9	$39, 57	9	$39, 57	9	code		

Cursor keys

INS	00 $52, 82	00 $4A, 74	00 $77, 119	Ins
⇧	00 $48, 72	00 $49, 73	00 $84, 132	Up, PgUp
DEL	00 $53, 83	00 $4E, 78	00 $75, 117	Del
⇦	00 $4B, 75	00 $47, 71	00 $73, 115	Left, Home
⇩	00 $50, 80	00 $51, 81	00 $76, 118	Down, PgDn
⇨	00 $4D, 77	00 $4F, 79	00 $74, 116	Right, End
CLS	$0C, 12	$0B, 11	$04, 4	
BS	$08, 8	$0F, 15	$7F, 127	BackSpace

Other keys

Func 1	00 $3B, 59	00 $54, 84	00 $68, 104	F1
Func 2	00 $3C, 60	00 $55, 85	00 $69, 105	F2
Func 3	00 $3D, 61	00 $56, 86	00 $6A, 106	F3
Func 4	00 $3E, 62	00 $57, 87	00 $6B, 107	F4
ETC.	00 $85, 133	00 $87, 135	00 $8B, 139	F11
MEMO	00 $3F, 63	00 $58, 88	00 $6C, 108	F5
IN	00 $40, 64	00 $59, 89	00 $6D, 109	F6
OUT	00 $41, 65	00 $5A, 90	00 $6E, 110	F7
CALC	00 $42, 66	00 $5B, 91	00 $6F, 111	F8
MENU	00 $43, 67	00 $5C, 92	00 $70, 112	F9
CAL	00 $44, 68	00 $5D, 93	00 $71, 113	F10
ON BRK	$03, 3	$03, 3	$03, 3	Break
EXE	$0D, 13	$0E, 14	$0A, 10	Enter
○ NEW ALL	00 $FF, 255	00 $FF, 255	00 $FF, 255	

Programmable Keys

The programmable keys ([S] *A* through [S] [SPC]) can be set to return an extended key code instead of the key sequence programmed at those keys. This is done to variable Crt.keyStatus, and is documented on page 256. In the table, the extended key codes are displayed.

The contents of the programmable keys can be changed in a Pascal program. The keys' contents are stored in memory at address $06E7 to $08C6. The easiest access to these keys are made by the variable declaration:

```
progKey :ARRAY [0..19] OF STRING[23]
            ABSOLUTE $06E7;
```

where the index in the array corresponds to the function key as seen on page 36.

The wanted key sequence is programmed by entering the characters wanted in the string. Double-code keys are entered as two characters: the null character (#00) followed by the key code (as seen in the tables).

The default contents of the programmable keys are restored when the **○ RESET** button is pressed, and when the ROM card is replaced.

Example:

To program key 0 ([S] *A*) to the text *FOR :=* followed by two cursor back codes (double-code $00,$4B), the following line would be used:

```
progKey[0]:='FOR :=' #0#$4B#0#$4B;
```

Cursor Keys

The cursor keys are located in the upper right of the keyboard. In addition to the cursor movement keys, the insert and delete keys are included here.

Other Keys

In addition to normal and cursor keys, there are the function keys, mode keys (such as **MENU** and **CAL** mode keys), and the **BREAK** key. In debugger mode, the [ON BRK] key has a special function, and will not return it's key code. To create the **BREAK** code in the debugger mode, press [S] [ON BRK].

Power-On Key Code

When the power switch is turned on, a special key sequence is entered into the key buffer. This is the double-code 00 $EF, 239. This code can be used e.g. to verify that all devices connected at the program start is still connected to the PB-2000.

Appendix D

Characters and Attributes

Two different character sets are built into the Pascal ROM card. The normal character set is the ANSI/ISO standard character set, based on the ISO 8859-1 standard. The other character set is the japanese character set. The character set used is specified by the current attribute flags.

In addition to the character set selection, the display can show each character as dimmed, reverse, and underline, by specifying the needed combination in Crt.textAttr.

There are also 64 user-programmable characters, at positions #00 to #31 and #128 to #159. These characters are pre-programmed with some national characters from the ISO character sets 8859-4 and 8859-6 (but at non-standard locations). User-programmable characters are restored to this default when the ○ **RESET** button is pressed, and when the ROM card is changed.

Character sets

The layout of the character sets are shown on the next page. All characters can be written to the display by using the Write procedure, except for characters in the range #07 to #13. For these characters, the cursor position will be changed when they are written, according to the following commands:

#07	Signal the beeper.
#08	Place the cursor one character to the left.
#09	Move cursor according to the current tabulator settings.
#10	Move the cursor one line down.
#11	Move the cursor to the top left corner of the window.
#12	Clear the window and move the cursor to the top left corner.
#13	Move the cursor to the beginning of the current line.

When directly set in the character display buffer, all characters can be displayed according to the character set selected.

ANSI/ISO character set

This character set is based on the ISO standard 8859-1. It also includes some characters from the 8859-4 and 8859-6 standards, but at non-standard locations. Those characters are located at #00 to #31 and #128 to #159, and can be programmed by the user.

	0x	1x	2x	3x	4x	5x	6x	7x	8x	9x	Ax	Bx	Cx	Dx	Ex	Fx
x0		Ř		0	@	P	`	p	Ą	Ő		°	À	Đ	à	đ
x1	〃	ř	!	1	A	Q	a	q	ą	ő	¡	±	Á	Ñ	á	ñ
x2	‶	Ľ	"	2	B	R	b	r	Ć	Ś	¢	²	Â	Ò	â	ò
x3	≈	ľ	#	3	C	S	c	s	ć	ś	£	³	Ã	Ó	ã	ó
x4	≡	Ň	$	4	D	T	d	t	Č	Š	¤	´	Ä	Ô	ä	ô
x5	π	ň	%	5	E	U	e	u	č	š	¥	µ	Å	Õ	å	õ
x6	κ	Ă	&	6	F	V	f	v	Ę	Ŧ	¦	¶	Æ	Ö	æ	ö
x7	⍾	ă	'	7	G	W	g	w	ę	ŧ	§	•	Ç	Œ	ç	œ
x8	α	b́	(	8	H	X	h	x	Ĳ	Ű	¨	¸	È	Ø	è	ø
x9	▸	ḿ	)	9	I	Y	i	y	ĳ	ű	©	¹	É	Ù	é	ù
xA	↓	ṕ	*	:	J	Z	j	z	Ł	Ź	ª	º	Ê	Ú	ê	ú
xB	↑	ẃ	+	;	K	[	k	{	ł	ź	«	»	Ë	Û	ë	û
xC	◤	ě	,	<	L	\	l	|	Ń	Ž	¬	¼	Ì	Ü	ì	ü
xD	◂	ů	-	=	M	]	m	}	ń	ž	⌐	½	Í	Y	í	y
xE	→	ĺ	.	>	N	^	n	~	Ŋ	Ż	®	¾	Î	Þ	î	þ
xF	←	ŕ	/	?	O	_	o		ŋ	ż	¯	¿	Ï	ß	ï	ÿ

Japanese character set

The japanese character set includes the normal katakana character set in normal locations, and also some graphic characters. The characters at the programmable locations are the same as for the ANSI/ISO character set.

	0x	1x	2x	3x	4x	5x	6x	7x	8x	9x	Ax	Bx	Cx	Dx	Ex	Fx
x0		Ř		0	@	P	`	p	Ą	Ő		ー	タ	ミ	═	╳
x1	〃	ř	!	1	A	Q	a	q	ą	ő	。	ア	チ	ム	╞	円
x2	‶	Ľ	"	2	B	R	b	r	Ć	Ś	「	イ	ツ	メ	╪	年
x3	≈	ľ	#	3	C	S	c	s	ć	ś	」	ウ	テ	モ	╡	月
x4	≡	Ň	$	4	D	T	d	t	Č	Š	、	エ	ト	ヤ	◢	日
x5	π	ň	%	5	E	U	e	u	č	š	・	オ	ナ	ユ	◣	時
x6	κ	Ă	&	6	F	V	f	v	Ę	Ŧ	ヲ	カ	ニ	ヨ	◥	分
x7	⍾	ă	'	7	G	W	g	w	ę	ŧ	ァ	キ	ヌ	ラ	◤	秒
x8	α	b́	(	8	H	X	h	x	Ĳ	Ű	ィ	ク	ネ	リ	♠	〒
x9	▸	ḿ	)	9	I	Y	i	y	ĳ	ű	ゥ	ケ	ノ	ル	♥	市
xA	↓	ṕ	*	:	J	Z	j	z	Ł	Ź	ェ	コ	ハ	レ	♦	区
xB	↑	ẃ	+	;	K	[	k	{	ł	ź	ォ	サ	ヒ	ロ	♣	町
xC	◤	ě	,	<	L	¥	l	|	Ń	Ž	ャ	シ	フ	ワ	●	村
xD	◂	ů	-	=	M	]	m	}	ń	ž	ュ	ス	ヘ	ン	○	人
xE	→	ĺ	.	>	N	^	n	~	Ŋ	Ż	ョ	セ	ホ	゛	╱	▒
xF	←	ŕ	/	?	O	_	o		ŋ	ż	ッ	ソ	マ	゜	╲	

Attributes

The default attribute setting is for normal characters. This can be changed by using variable Crt.textAttr. Any combination of underline, reverse, and dimmed attributes can be set. To change the attribute setting, set the variable to the sum of the wanted combination of attributes, or to zero to use the normal ANSI/ISO character set.

The underline attribute is specified by value 1. The characters will then be underlined.

Dimmed attribute is set by value 2. With this setting, some pixels in the character will be off, to make the impression that the character has a lower intensity level than normally.

Characters can be reversed by using value 4 for the attribute. All pixels in the character will be reversed.

Last, the japanese character set is specified by the value 128.

Character Display Buffer

To make output to the display faster, the character display buffer can be directly accessed. This buffer is located at address $0100. The buffer consists of 4 lines, each of 32 characters and attributes in an interleaved manner. The top left character is set at address $0100, the attribute for this character at address $0101, the next character at address $0102, and so on. The declaration for the display buffer could be made as:

```
charDisp : ARRAY [0..3,0..31,0..1] OF CHAR
              ABSOLUTE $0100;
```

where the top left character is at charDisp[0,0,0], the attribute at charDisp[0,0,1], and the next character on the same line at charDisp[0,1,0].

After updating the display buffer, a call to Crt.TextDisplay is needed to convert the characters to graphic data, and to update the LCD display. Only the attributes set in the display buffer will be used for conversion. The variable Crt.textAttr is not used by the Crt.TextDisplay procedure.

Example:

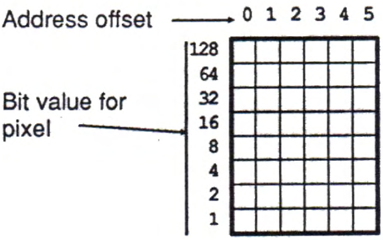
The following program will show the last 128 character of the ANSI/ISO character set on the display. The display buffer is here set as a simple Word array, to simplify the program. Both the character and the attribute to set is specified by the variable ch.

```
PROGRAM FastDisplay;
USES Crt;
VAR
  ch,pos : Word;
  charDisp : ARRAY [0..127] OF Word ABSOLUTE $0100;
BEGIN
  FOR ch:=128 TO 255 DO BEGIN
    FOR pos:=0 TO 127 DO charDisp[pos]:=ch;
    TextDisplay;
  END;
  ClrScr;
END.
```

Programmable Characters

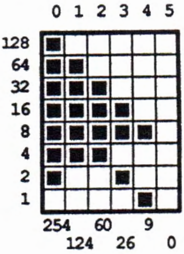
The programmable characters are set starting at address $0500. Each character occupies six bytes, where the upper left pixel is the highest bit (value 128) of the first byte. The table to the right shows the correspondence between the character bit and the address.

The starting address for each character is according to the following table:



#00	$0500	#16	$0560	#128	$05C0	#144	$0620
#01	$0506	#17	$0566	#129	$05C6	#145	$0626
#02	$050C	#18	$056C	#130	$05CC	#146	$062C
#03	$0512	#19	$0572	#131	$05D2	#147	$0632
#04	$0518	#20	$0578	#132	$05D8	#148	$0638
#05	$051E	#21	$057E	#133	$05DE	#149	$063E
#06	$0524	#22	$0584	#134	$05E4	#150	$0644
#07	$052A	#23	$058A	#135	$05EA	#151	$064A
#08	$0530	#24	$0590	#136	$05F0	#152	$0650
#09	$0536	#25	$0596	#137	$05F6	#153	$0656
#10	$053C	#26	$059C	#138	$05FC	#154	$065C
#11	$0542	#27	$05A2	#139	$0602	#155	$0662
#12	$0548	#28	$05A8	#140	$0608	#156	$0668
#13	$054E	#29	$05AE	#141	$060E	#157	$066E
#14	$0554	#30	$05B4	#142	$0614	#158	$0674
#15	$055A	#31	$05BA	#143	$061A	#159	$067A

Example:



The special character to the right is to be programmed at character position #04. For this purpose, the sum of each column in the character is calculated. The character is most easily set as an initialized variable, which is then copied to the programmable character at the beginning of the program.

```
PROGRAM OwnChar;
TYPE
  CharType = ARRAY [1..6] OF Byte;
VAR
  genChar : CharType ABSOLUTE $0518;
  newChar : CharType = (254,124,60,26,9,0);
BEGIN
  genChar:=newChar;
  Write (#4);
END.
```

Appendix E

Error Messages

When an error is detected, an error message is written. This message specifies the type of the error.

Normally when a program is being compiled, the editor is automatically called when a compilation error is detected. When compiling from disk, however, this cannot be done since the source file is on the disk. In this case, an error message is written, specifying the file where the error occurred, together with the source line number.

An error occurring while a program is executing is called a *run-time* error. If debug information can be found at the position of the run-time error, both the error message, the current file being executed, and the line number at the error position is written. Without debug information, only the error message is written.

The next page shows the error numbers returned by the IOResult function (in programs) and ERR variable (in **CAL** mode). The errors are described below in alphabetic order, with the corresponding error number shown in parentheses.

Run-time Error Codes

The following run-time error codes can be set.

Access denied (5)

Access to the specified file is denied. This is normally the case when a file already exists that a function (e.g. Rename) does not expect.

Argument error (51)

The argument to the function is not within the allowed range, e.g. Sqrt(–2).

Run-time Error Codes

1	Battery error
2	Not found error
3	Protect error
4	Too many files
5	Access denied
6	File handle error
14	Format error
15	Device error
16	Name error
17	Rename error
18	No more files
20	Read fault
21	Write fault
22	Not ready error
23	Command error
30	Read error
31	Write error
33	Open error
34	Input error
35	Output error
36	Numeric error
50	Real overflow
51	Argument error
52	Range error
53	Pointer error
54	Division by zero
55	Stack overflow
56	Heap overflow
80	Out of memory
81	Line too long
82	Syntax error
83	User break
90	Unit not found
91	Version error
92	Relocate error

Compile Error Codes

100	Real constant error
101	Integer constant error
102	String past line end
103	Nested file error
104	Ident.not defined
105	Ident.redefined
106	EOF not expected
107	Directive expected
108	IFxxx missing
109	Ident.expected
110	Label undefined
111	Pointer type missing
112	Proc/func body missing
113	Assignment error
114	Subrange type error
118	Type not expected
119	Too many WITH nested
120	Must be VAR declared
121	Type id expected
122	Return type error
123	ConstExpr error
126	Const range error
127	Record field error
128	Type mismatch
129	Too many variables
130	’OF’ expected
131	’:’ expected
132	’;’ expected
133	’END’ expected
134	’:=’ expected
135	’BEGIN’ expected
136	’TO/DOWNTO’ expected
137	’THEN’ expected
138	’DO’ expected
139	’UNTIL’ expected
140	’)’ expected
141	’(’ expected
142	’,’ expected
143	’=’ expected
144	’]’ expected
145	’[’ expected
146	’..’ expected
147	’.’ expected
150	Label level error
151	Label expected
152	Expr too complex
153	Illegal operator
154	Redeclare error
155	Long pointer error

Battery error (1)

The batteries in the MD-100 option unit has been discharged too much for the computer to be able to use the floppy disk. Replace the batteries to use the MD-100.

Command error (23)

An illegal command has been sent to the MD-100's floppy disk.

Device error (15)

The device used is not a correct device name, not allowed with this function, or is currently not connected to the PB-2000.

Division by zero (54)

A real or integer division by zero has been attempted.

File handle error (6)

The file handle sent to a file function is not a valid file handle.

Format error (14)

The floppy disk has not been formatted, or the disk contents have been damaged.

Heap overflow (56)

The heap area has grown too large, either due to too many global variables, or too many dynamically allocated variables.

Input error (34)

The file used has only been opened for output by Rewrite.

Line too long (81)

The **CAL** line to calculate is longer than 255 characters. The cursor is placed at the 256'th character.

Name error (16)

The same file specifier has been entered to the COPY or APPEND functions, both for the source and the destination file specifiers.

No more files (18)

This is no normal error, but specifies that no more files can be found by a directory command.

Not found error (2)

The file with the specified file name cannot be found. Make sure the upper and lower casing of the file name is correct.

Not ready error (22)

The device used cannot be read from, or the ON BRK key was pressed while waiting for the device. This can be due to the handshake lines being in inactive state, or the floppy disk not inserted into the floppy disk drive.

Numeric error (36)

This error is set when reading a real or integer variable, and the read string does not comply to the needed format.

Open error (33)

The file used has not been opened by Reset or Rewrite.

Out of memory (80)

There are not enough memory for the wanted setting for CLEAR command or `[memory]` function key. The error can also be set when there is not enough space for a new variable in the CAL area.

Output error (35)

The file used has only been opened for input by Reset.

Pointer error (53)

The pointer variable being dereferenced has currently the value NIL.

Protect error (3)

The floppy disk has the write-allow notch uncovered (set in the write-protect state).

Range error (52)

A subrange variable is being assigned a value too small or large.

Read error (30)

A read error has occurred during execution of Read.

Read fault (20)

Reading from the device detected an illegal character being received.

Real overflow (50)

The result of a real operation has exceeded the allowed range for the Real variable. Note that the range must not be exceeded, even for intermediate calculations.

Relocate error:fileName (92)

The relocation table of the file fileName has been corrupted.

Rename error (17)

The specified devices for the old and new name in the Rename command are different.

Stack overflow (55)

The stack has grown too large, either due to too many recursive calls, or too many local or parameter variables. The setup of the memory areas might need to be changed.

Syntax error (82)

The expression being calculated cannot be decoded at the cursor position.

Too many files (4)

There are too many files opened at this moment. A maximum of 20 files can be opened at one time.

Unit not found:fileName (90)

The file fileName cannot be found, but is needed for the file currently being compiled or executed.

User break (83)

The ON BRK key has been pressed during execution of a file command.

Version error:fileName (91)

The file importing fileName has another version or checksum number than this file. This error can be set if the file has been corrupted, or not being recompiled after changing an interface file.

Write error (31)

An error has occurred during execution of Write.

Write fault (21)

Writing to the device cannot be made due to full memory or disk.

Compile Error Codes

The following errors can be written at compile time, in addition to the run-time errors.

'.' expected (147)

A period was expected at this place in the program.

'..' expected (146)

A subrange specifier was expected when used at this position.

',' expected (142)

A comma was expected at this point. Often this error signals that too few parameters were entered to a procedure or function.

'=' expected (143)

An equal sign was expected, e.g. in CONST and TYPE declarations.

':' expected (131)

A colon was expected, e.g. at declaration of variables.

':=' expected (134)

An assignment symbol was expected after the variable in the statement.

';' expected (132)

A semicolon was expected at this position in the program. This error might be set on the 'wrong' line; the semicolon is normally written at the end of the statement of the previous line.

'(' expected (141)

A left parenthesis is expected. This error is set if parameters are to be used with a subroutine, but no one has been entered.

')' expected (140)

A right parenthesis was expected. If more parameters than allowed for a procedure or function are entered, this error is set.

'[' expected (145)

A left square bracket was expected, e.g. after the ARRAY keyword at array type declarations.

']' expected (144)

A right square bracket was expected at this position, e.g. after array index declarations.

Assignment error (113)

Assignment is not allowed to an identifier of this type.

'BEGIN' expected (135)

The BEGIN keyword must be entered to start the main part of the program or subroutine. This error can also be set if a non-allowed keyword is found while decoding the declarations.

ConstExpr error (123)

The expression entered has types or operators not allowed in a constant expression.

Const range error (126)

The constant value entered is too small or large for the allowed range of values in the identifier.

Directive expected (107)

A valid compiler directive was expected after the leading {$ or (*$ symbols.

'DO' expected (138)

The DO keyword was expected as the next symbol in a WHILE or WITH statement.

'END' expected (133)

The END keyword was expected to end the CASE statement or BEGIN block.

EOF not expected (106)

The end of the source file was found, but more symbols was expected before the end of the program is reached. This error can be set if a non-terminated comment has been found in the program.

Expr too complex (152)

The expression or parameter call list is too complex to fit into the compiler's syntax tree. Split the expression into smaller parts to get around this error.

Heap overflow (56)

The heap is too small for the symbol table and temporary variables used when compiling this program. Increase the heap size to be able to compile the program.

Ident.expected (109)

An identifier is expected at this position in the program.

Ident.not defined (104)

An identifier is used before it has been defined in the program.

Ident.redefined (105)

The identifier has already been used at this subroutine level.

IFxxx missing (108)

The compiler directive ELSE or ENDIF is used without a matching IF compiler directive.

Illegal operator (153)

The operator used is not allowed with the arguments entered.

Integer constant error (101)

An illegal character was found while decoding an integer, or the resulting integer value is too large.

Label expected (151)

The identifier used in the GOTO statement must be a label identifier.

Label level error (150)

The label being used is declared at another subroutine level, and cannot be used at this level.

Label undefined (110)

A label has been declared at the current subroutine level, but not been used in the subroutine.

Long pointer error (155)

A variable declared as a long pointer, or not in bank number 1, is not allowed at this position, e.g. as a VAR-declared parameter.

Must be VAR declared (120)

An open array must be declared as a variable-type parameter.

Nested file error (103)

No more than three include files can be nested.

'OF' expected (130)

The OF keyword was expected in this context.

Out of memory (80)

There is too little memory to hold the resulting code file. Increase the memory by reducing the heap or removing other files.

Pointer type missing (111)

A pointer has been declared, with the base type not declared at the same level.

Proc/func body missing (112)

A forward-declared procedure or function has not been fully declared.

Real constant error (100)

An illegal character was found while decoding a real number, or the resulting real value is too large.

Record field error (127)

An illegal symbol was found in a record field declaration.

Redeclare error (154)

The redeclared subroutine header was different from the one used at the forward-declaration, or the one used in the interface unit.

Return type error (122)

This type is not allowed for the return type of a function.

String past line end (102)

The string constant being decoded was not terminated before the end of the line.

Subrange type error (114)

A subrange type was expected at this position in the program.

'THEN' expected (137)

The THEN keyword is expected in the IF statement.

'TO/DOWNTO' expected (136)

Either the TO or DOWNTO keyword is expected in the FOR statement.

Too many variables (129)

There are too much memory used by the variables at this subroutine level. Reduce the number of variables, or place some variables at the global data level.

Too many WITH nested (119)

A maximum of 10 nested WITH statements are allowed.

Type id expected (121)

At this position in the program only a type identifier is allowed.

Type mismatch (128)

The type of the variable used does not match the parameter type declared.

Type not expected (118)

This identifier type is not expected at this position in the program.

'UNTIL' expected (139)

The UNTIL keyword is expected at the end of the REPEAT statement.

User break (83)

The ON BRK key was pressed during compilation, or the power switch was turned off.

Appendix F

System Memory Area

The system memory area is split into three parts: the *heap memory*, the *stack memory*, and the *numeric variable area*. They have different usage in **CAL** mode, compile time, and at program execution.

The normal RAM area of the PB-2000 is located in bank 1. The ROM of the compiler is located in bank 0. Normal variables (short pointers) are always located in bank 1, but by declaring absolute variables, or by using long pointers, they can also be located in bank 0.

All of these memory areas can be changed either by the `[memory]` function key in **MENU** mode, or by the CLEAR command in **CAL** mode.

Heap Memory Area

The heap is used to hold the **CAL** area in the **CAL** mode.

At program compilation, the heap is used to store some variables, and the symbol table being built by the compiler.

When a program is being run, the heap holds all global variables and the dynamically allocated variables. Also, parameters and local variables are stored in the high end of the heap.

The heap area must have a size of at least 2 Kbytes.

Stack Memory Area

In all modes, the stack is used to store the return address when calling subroutines.

In addition, most of the area is used at compilation time to hold the conditional compilation identifiers being declared.

The stack area must be allocated a size of at least 512 bytes.

Numeric Variable Area

The numeric variable area is only used to store the variables used in **CAL** mode. Even at compilation and program execution, this area is not changed.

To delete the contents of this area, the CLEAR command in **CAL** mode should be used, without any parameters.

This area can only be allocated a size of more than 256 bytes.

System Memory

The following addresses in the system memory are of special interest to a programmer. The data at other addresses are used by the system for internal purposes, and should not be changed.

$0100-01FF	Character display buffer. This buffer is documented on page 299.
$0200-04FF	Graphic display buffer. This buffer is documented on page 213.
$0500-067F	Programmable character set. The structure and use of these characters are described on page 300.
$0680	Current display attribute. The variable Crt.textAttr points to this memory address.
$0681	Default display attribute.
$0682	Display status flags. Bit 0 is cursor on flag, and bit 1 is set when the cursor is really visible.
$0683	User display flags. Variable Crt.dispStatus uses this memory address.
$0684	Left x value for update of the LCD screen. Set to 32 when no update is needed.
$0685	Right x value for update of the LCD screen. Set to 0 when no update is needed.
$0686	Upper y value for LCD screen update. Set to 3 for no update.
$0687	Lower y value for LCD screen update. When no update is needed, this is set to 0.
$0688	Current x position for text output. 0 is leftmost position in the screen.
$0689	Current y position for text output. 0 is the top line in the window.
$068A	Left corner of the window.
$068B	Right corner of the window.

$068C	Top line of the window.
$068D	Bottom line of the window.
$068E-98	Last real random number, used as the base for the next random number.
$06A9-AA	Current time of day, in minutes. The value is set to 0 at midnight. This value is only updated as long as the computer is on. The seconds can be read in the TM register.
$06AB-AC	Current day, as counted from 1980-01-01. This value is only updated as long as the computer is on.
$06E7-08C6	Key buffer for the programmable keys. This is described on page 294.
$08D8	Last language code sent when translating output for Epson printers.
$08D9	Current value of the auto power-off timer.
$08DA	Default value of the auto power-off timer, to be used when waiting for a key press.
$08E0	Current number of characters in the RS-232C receive buffer.
$08E1	Current write offset in the RS-232C receive buffer.
$08E2-09E1	RS-232C receive buffer.
$09E4	Key shift and toggle status. Bit 0 is insert status, bit 3 pending contrast setting, bit 4 ans, bit 5 kana toggle, bit 6 caps toggle, and bit 7 shift status flag.
$09E5	Current key status. This address is used by the Crt.keyStatus variable.
$09EC	Current number of characters in keyboard buffer.
$09ED	Last read position in the keyboard buffer.
$09EE	Last write position in the keyboard buffer.
$09EF	Size of the keyboard buffer.
$09F0-F1	Address to the keyboard buffer.
$0A03	Current trigonometric angle flag. It works the same way as the angle variable.
$0A05	Current tabulation setting. Tab stops are set at this distance from the left margin of the screen and from each other.
$0A06-45	Keyboard buffer (64 characters).
$0AA0-A1	Pointer to first entry in free area list of dynamic memory, or zero if no free list exists.
$0AA2-A3	Start address of the heap. Always set to $1800.

$0AA4-A5	Pointer to the first free byte on the heap.
$0AA6-A7	Initial value of the user stack pointer.
$0AA8-A9	Initial value of the system stack pointer.
$0AAA-AB	Pointer to the first data byte used by **EXE**, **DLE**, and **UNI** files.
$0AAC-AD	Pointer to the first data byte of other files.
$0AAE-AF	Pointer to first free data byte after the files.
$0AB0-B1	Pointer to last file information block.
$0AB2-B3	Pointer to last **EXE**, **DLE**, and **UNI** file information block.
$0AB4-B5	End of memory address ($7FFF for 32 KByte, $FFFF for 64 KByte).
$0ABA-BB	Printer flag error test before and after output of character. Bit 0 is busy test, bit 1 is error test, and bit 2 is ack test.
$0ABC	Flag for connected interface unit. Value $FF specifies no unit, $00 specifies FA-7, and $55 specifies MD-100.
$0ABD	File handling flags. Set value 1 if RS-232C and printer can time-out (instead of wait until ready). Set value 2 if file names read from the cassette recorder are not to be written on the display.
$0ABE	Type of CAS or COM device opened. This is 0 when none is opened, 2 for COM, and 6 for CAS device.
$0ABF	Number of CAS or COM files opened.
$0AC0-C7	The last calculated **CAL** value as a Real type.
$0F00-11FF	FAT information of floppy disk.
$1200	FAT information has been read.
$1201	FAT information has been changed.
$1202	Disk buffer has information.
$1203-04	Disk sector loaded in disk buffer.
$1205	Disk buffer has been changed.
$1206-1305	Disk buffer.
$142E-152F	Parameter area, as entered when calling a program from the **CAL** mode. The line is stripped of redirection specifications, and any batch parameter entered has been expanded.
$1530-162F	Temporary buffer, used internally by various functions and procedures, but otherwise free to be used by a program.
$1630-1719	System unit data. This should only be updated from the variables, procedures, and functions in the System unit.
$171A-AF	Free area in the system memory area.

Appendix G

CPU Instruction Codes

This appendix contains a summary of the CPU instruction codes for the PB-2000, for use in the INLINE statement. For more thorough description of these assembler codes, you should look for a book teaching assembler, especially for the PB-1000 or PB-2000 series computers.

All values given in this appendix are hexadecimal values. Values prefixed by a dollar sign are **not** hexadecimal values; they are the CPU registers. For the operation code operands, some abbreviations are used. They are the following:

c	direction flag: + (00) or – (80)
co	condition: Z (0), NC (1), LZ (2), UZ (3), NZ (4), C (5)
d	destination register: 00 through 1F
ir	index register: IX (0) and IZ (1)
m	word-size constant; ml is low byte, mh is high byte
n	byte-size constant: 00 through FF
r	normal register: 00 through 1F
s	source register: 00 through 1F
t1	status register: PE (00), PD (20), UA (60)
t2	status register: IA (00), IE (20), TM (60)
w1	word register: IX (00), IY (20), IZ (40), US (60)
w2	word register: SS (00), KY (60)

Example:

To make a function return with it's parameter multiplied by ten, the following assembler program is written:

```
LD     $4,4
LDW    $8,(IX+$31)
LDW    $6,$8
BIUW   $8
BIUW   $8
ADW    $8,$6
BIUW   $8
STW    $8,(IX-$4)
```

AD (ir±n),s	7C+ir s+c n
AD (ir±r),s	3C+ir 60+s+c r
AD d,n	48 d n
AD d,s	08 60+d s
ADB d,n	4A d n
ADB d,s	0A 60+d s
ADBW d,s	8A 60+d s
ADC (ir±n),s	78+ir s+c n
ADC (ir±r),s	38+ir 60+s+c r
ADC d,n	40 d n
ADC d,s	00 60+d s
ADCW (ir±r),s	B8+ir 60+s+c r
ADCW d,s	80 60+d s
ADW (ir±r),s	BC+ir 60+s+c r
ADW d,s	88 60+d s
AN d,n	4C d n
AN d,s	0C 60+d s
ANC d,n	44 d n
ANC d,s	04 60+d s
ANCW d,s	84 60+d s
ANW d,s	8C 60+d s
BDN	D9
BID d	18 40+d
BIDW d	98 40+d
BIU d	18 60+d
BIUW d	98 60+d
BUP	D8
BYDW d	9A 40+d
BYUW d	9A 60+d
CAL co,m	70+co ml mh
CAL m	77 ml mh
CANI	FC
CLT	F9
CMP d	1B d
CMPW d	9B d
DID d	1A d
DIDW d	9A d
DIU d	1A 20+d
DIUW d	9A 20+d
FST	FA
GFL d	1C 40+d
GPO d	1C d
GRE w1,d	9E w1+d
GRE w2,d	9F w2+d
GST t1,d	1E t1+d
GST t2,d	1F t2+d
INV d	1B 40+d
INVW d	9B 40+d
JP co,m	30+co ml mh
JP m	37 ml mh
JR co,n	B0+co n
JR n	B7 n
LD d,(ir±n)	68+ir 60+d+c n
LD d,(ir±r)	28+ir 60+d+c r
LD d,(s)	11 60+d s
LD d,n	42 d n
LD d,s	02 60+d s
LDI d,(ir±n)	6A+ir 60+d+c n
LDI d,(ir±r)	2A+ir 60+d+c r
LDIW d,(ir±r)	AA+ir 60+d+c r
LDW d,(ir±r)	A8+ir 60+d+c r
LDW d,(s)	91 60+d s
LDW d,m	D1 d ml mh
LDW d,s	82 60+d s
NA d,n	4D d n
NA d,s	0D 60+d s
NAC d,n	45 d n
NAC d,s	05 60+d s
NACW d,s	85 60+d s
NAW d,s	8D 60+d s
NOP	F8
OFF	FE
OR d,n	4E d n
OR d,s	0E 60+d s
ORC d,n	46 d n
ORC d,s	06 60+d s
ORCW d,s	86 60+d s
ORW d,s	8E 60+d s

PFL s	14 40+s
PHS s	26 s
PHSW s	A6 s
PHU s	27 s
PHUW s	A7 s
PPS d	2E d
PPSW d	AE d
PPU d	2F d
PPUW d	AF d
PRE w1,m	D6 w1 ml mh
PRE w1,s	96 w1+d
PRE w2,m	D7 w2 ml mh
PRE w2,s	97 w2+d
PST t1,n	56 t1 n
PST t1,s	16 t1+s
PST t2,n	57 t2 n
PST t2,s	17 t2+s
ROD d	18 d
RODW d	98 d
ROU d	18 20+d
ROUW d	98 20+d
RTN	F7
RTN co	F0+co
RTNI	FD
SB (ir±n),s	7E+ir s+c n
SB (ir±r),s	3E+ir 60+s+c r
SB d,n	49 d n
SB d,s	09 60+d s
SBB d,n	4B d n
SBB d,s	0B 60+d s
SBBW d,s	8B 60+d s
SBC (ir±n),s	7A+ir s+c n
SBC (ir±r),s	3A+ir 60+s+c r
SBC d,n	41 d n
SBC d,s	01 60+d s
SBCW (ir±r),s	BA+ir 60+s+c r
SBCW d,s	81 60+d s
SBW (ir±r),s	BE+ir 60+s+c r
SBW d,s	89 60+d s
SDN n	5D n
SDN r	DD r
SLW	FB
ST s,(d)	10 60+s d
ST s,(ir±n)	60+ir 60+s+c n
ST s,(ir±r)	20+ir 60+s+c r
STI s,(ir±n)	62+ir 60+s+c n
STI s,(ir±r)	22+ir 60+s+c r
STIW s,(ir±r)	A2+ir 60+s+c r
STW s,(d)	90 60+s d
STW s,(ir±r)	A0+ir 60+s+c r
SUP n	5C n
SUP r	DC r
TRP m	FF ml mh
XR d,n	4F d n
XR d,s	0F 60+d s
XRC d,n	47 d n
XRC d,s	07 60+d s
XRCW d,s	87 60+d s
XRW d,s	8F 60+d s

Look up the operation codes corresponding to these assembler codes. E.g. for the instruction `LDW $6,$8`, the table says: 82 60+d s. Replace s by the source register (8), and d by the destination register (6). The resulting three-byte code will be: 82 66 08. When this has been done for all instructions, the result should be:

```
LD     $4,4               42 04 04
LDW    $8,(IX+$31)        A8 68 1F
LDW    $6,$8              82 66 08
BIUW   $8                 98 68
BIUW   $8                 98 68
ADW    $8,$6              88 68 06
BIUW   $8                 98 68
STW    $8,(IX-$4)         A0 E8 04
```

Now write the function with inline code, using these instruction codes.

```
FUNCTION Mul10 (w:Word) :Word;
  BEGIN
    INLINE ($42,$04,$04, $A8,$68,$1F, $82,$66,$08,
            $98,$68, $98,$68, $88,$68,$06, $98,$68,
            $A0,$E8,$04);
  END;
```

Index

!

#number 6
$number 6
0: device 38

A

A: device 38
Abs function 218
  CAL mode 59
Absolute variable 147
AC Adapter
  interface 110
  PB-2000 287
Addr function 218
Address
  function Addr 218
  operator 168
  raw window, debugger 106
  reference 169
Alias of type 135
Allocation, dynamic memory . . 189
Alternate shift key 8
ANGLE command 62
angle variable 216
ANS key 8
ANSI character set 297
APPEND command 62
Append file 48
append function key 48
Append procedure 218
ArcCos function 219
  CAL mode 59
ArcCosH function 219
  CAL mode 59
ArcSin function 219
  CAL mode 59
ArcSinH function 219
  CAL mode 59
ArcTan function 220
  CAL mode 59
ArcTanH function 220
  CAL mode 59
Argument 166
  value 167
Arithmetic operator 160, 166
Array
  dereference 170
  element 142
  initialized variable 148
  multi-dimensional 142
  open parameter 154
  type 142
  type format 274
Ask at replace 80
Assign file variable 195
Assign procedure 220
AssignCrt procedure 257
Assignment compatibility . . . 145
Assignment statement 183
Attribute, display 299
Automatic indentation 87
Automatic power-off 52
AUX: device 38

B

Backup battery 285
Bank, memory 273
Base type
  pointer 138
  set 143
  subrange 137
BASIC 109
.BAT extension 38

Batch file 45, 61, 66, 71
Battery
  change PB-2000 285
  interface 110
  PB-2000 285
BEEP command 62
Beeper, key 52, 62
`begblk` function key 85
`begin` function key 81
BENCHCAL.PAS 32
BENCHFOR.PAS 30
BENCHTEX.INT 28
BENCHTEX.PAS 29
Binary operator 160
Block 80
  copy 82
  delete 82
  goto beginning 85
  goto end 85
  hide 81
  mark 80
  move 82
  save 83
  set beginning 81
  set end 81
  undo deletion 81
  unhide 81
`block` function key 80
BlockRead procedure 221
BlockWrite procedure 222
Body
  program 129
  subroutine 155
Boolean
  operator 162, 167
  short-circuit 163
Boolean type 215
Break key 10
Break point 99
  clear 102
  set 102
BRK key 10, 294
BS key 9
Buffer
  character display 299
  graphic 213
Button
  NEW ALL 11
  RESET 11
Byte type 215

C

C compiler flag 268
C: device 38
Cable for RS-232 113
CAL key 9
CAL mode 55
  clear variable 58
CALC key 10
Calculation memory 93
Calculator 57
Call
  function 172
  procedure 183
CAPS key 8
Card, ROM 109, 289
  change 289
  start 290
CAS: device 38
Case
  CAPS key 8
  identifier casing 122
  insensitive search 80
CASE statement 177
  record variant 140
Case, hard for PB-2000 111
Cassette interface 114
  speed 114
Casting
  See Conversion
CD control, RS-232 41
Ceil function 222
  CAL mode 59
Centronics interface 113
Change
  battery PB-2000 285
  disk 54, 115
  ROM card 289
Char type 215

Character
  attribute display 299
  control 297
  conversion 39, 41, 64
  display buffer 299
  IBM set 49
  number 6
  operator 164
  Pascal 124
  programmable 300
  set 297
  type 136
Checksum, unit 27
Chr function 222
Circular import 187
Clear
  break point 102
  CAL variable 58
CLEAR command 62
Close
  file 195
  user text type 202
Close procedure 223
`clr bp` function key 102
ClrEol procedure 258
ClrScr procedure 258
CLS command 63
CLS key 9
`cnv fr` function key 49
`cnv to` function key 49
Code
  file 15
  key 291
  window, debugger 100
`code` function key 105, 107
Colon, EBNF 119
COM: device 38
Command line 61
Comment 125
Communication protocol 39
COMP command 63
Compare
  CAL mode 58
  Pascal 279

Compatibility
  assignment 145
  expression 146
  type 144
  type identity 145
`compil` function key . 46, 54, 84
Compile 46
  conditional 269
  disk file 54
  editor 84
  error 309
  file 63
  memory setting 271
  separate 27
Compiler 3, 97
  directive 271
  flag 267
  flag test 271
  option 267
Compound statement 175
CON: device 38
Concat function 223
Concat string operator 164
Conditional compilation 269
Conditional identifier 269
Constant
  declaration 133
  expression 160
Contrast setting 11
Control
  character 297
  element, CASE 177
  expression, CASE 177
  variable, FOR 179
Conversion
  character 39, 41, 64
  file 49
  type 169
CONVFR command 64
CONVTO command 64
Coordinate
  graphic 211
  text 209

Copy
  block 82
  file 64
  subroutine flag 268
  value parameter 152
COPY command 64
Copy function 224
copy function key 82
Cos function 224
  CAL mode 59
CosH function 224
  CAL mode 59
CPU instructions 321
Crt unit 255
CTS control, RS-232 41
Curly brace, EBNF 120
curr function key 107
Cursor 7
  CAL mode 55
  down key 9
  editor 76
  left key 9
  line edit 36
  MENU mode 43
  move in window 210
  prev position 85
  right key 9
  up key 9
Cursor procedure 259

D

D compiler flag 268
Data bits, RS-232 40
data function key 104, 107
Data window, debugger 104
Dead key 291
Debug 99
  goto current pos 107
  goto file 107
  information 99, 268
  information, code 15
  program 45, 65
DEBUG command 65
Dec procedure 224
Declaration 128
  constant 133
  label 131
  subroutine 151
  type 135
  uses unit 128
  variable 147
DEFINE compiler option 270
Definition line, EBNF 119
DEFKEY command 65
DEL command 66
DEL key 9
Delete
  block 82
  disk file 54
  file 47, 66, 195
  undo block 81
delete function key 82
Delete procedure 225
DelLine procedure 259
Dereference 170
  array 170
  pointer 171
  pointer NIL test 269
  record 171
  string 170
Device, file 37
Dimmed menu 8
DIR command 66
Direction in FOR loop 179
Disk
  compile file 54
  delete file 54
  format 67
  initialize 67
  interface 114
  load file 54
  MENU 46, 53
  rename file 54
  replace 54
disk function key 46, 53
Display 7
  character buffer 299
  graphic buffer 213
Display procedure 211, 260
Dispose procedure 191, 225
dispStatus variable 255

DL-Pascal 4
.DLE extension 38
DO command 66
Double-character symbol 125
Double-key code 291
DOWNTO keyword 179
DSR control, RS-232 41
Dynamic
  linking 185
  memory 189
  variable 189

E

E file type 39
EBNF 119
ECHO command 72
ECHOX command 72
Edit
  file 45, 67
  input line 36
  new file 45
EDIT command 67
`edit` function key 45
Editor 75
  full-screen 87
  indentation 87
  user features 86
  wrap-around line 87
Element
  array 142
  CASE control 177
  enumerated 137
  file type 144
Ellipses 6
ELSE compiler option 271
ELSE keyword
  in CASE 177
  in IF 176
Empty
  set 167
  statement 175
`end` function key 81
`endblk` function key 85
ENDIF compiler option 271
Enter
  raw window data 106
  variable data 105

`enter` function key . . . 105-106
Enumeration
  operator 162
  type 137
Eof function 226
Eoln function 226
Erase
  See Delete
Erase procedure 226
Err function
  CAL mode 60
Error
  compile 309
  file handling 195
  file test 268
  message 303
  range test 269
  run-time 303
errorAddr variable 207, 216
Exchange local/global 104
.EXE extension 38
EXE key 10, 13
  MENU 45
Execute
  debug program 45, 65
  goto current pos 107
  program 45, 68
  set new pos 103
  speed 99
  terminate 207
  to cursor pos 103
  to end 102
  to next statement 101
Exit procedure 227
exitCode variable 207, 216
ExitProc variable 208, 216
Exp function 227
  CAL mode 60
Expression 159
  CASE control 177
  compatibility 146
  constant 160
  sub- 167
Extended key code 291
Extension, file 37-38

F

- F compiler flag 268
- FA-7 interface 110
- False value 137, 215
- Far pointer 138
- Field in record 140
- File
  - append 48
  - assign 195
  - batch 45, 61, 66, 71
  - close 195
  - code 15
  - compile 54, 63, 84
  - conversion 49
  - copy 64
  - delete 47, 54, 66, 195
  - device 37
  - edit 45, 67
  - edit new 45
  - element type 144
  - error handling 195
  - error test 268
  - extension 37-38
  - goto, debugger 107
  - implementation 27
  - include compiler 271
  - insert in editor 84
  - interface 27
  - list 66
  - load 47
  - load disk 54
  - MENU group select 44
  - name 37-38
  - open 195
  - preset 24, 53, 67
  - print 50
  - printer name 39
  - redirection 61
  - rename 46, 54, 68, 195
  - RS-232 name 39
  - save 48
  - source 15
  - specifier 37
  - text 144, 198
  - type 39, 143
  - type format 275

File *(continued)*

  - typed 144, 196
  - unit name 38
  - untyped 144, 197
  - variable 195
  - verify 69
- `file` function key 107
- FilePos function 228
- FileSize function 228
- FillChar procedure 228
- FIRST.PAS 16
- Flag, compiler 267
- `flags` function key 79
- Floor function 229
  - CAL mode 60
- Floppy disk 114
  - insert 115
  - remove 115
- FOR statement 179
- FORMAT command 67
- Format disk 67
- Format of variable 273
- Forward declared subroutine 156
- FORWARD keyword 156
- Frac function 229
  - CAL mode 60
- Free memory 51
- Free-format language 121
- FreeMem procedure 191, 229
- Full-screen display, editor 87
- Function
  - body 155
  - call 172
  - declaration 151
  - forward 156
  - heading 151
  - level 155
  - main part 155
  - parameter 152
  - return type 151
  - type 140
  - type format 274
- Function key 8
  - debug 100
  - editor 75
  - MENU mode 43

- Function memory 93
  - MEMO mode 95
- Function menu 8

G

- `g.here` function key 103
- GetMem procedure 190, 229
- Global
  - identifier 129
  - identifier, prefix 155
  - replace, editor 80
  - variable 147
  - variable, debugger 104
- `go end` function key 102
- `go nxt` function key 101
- Goto 84
  - block beginning 85
  - block end 85
  - current pos, debugger 107
  - file, debugger 107
  - line number 85
  - prev cursor 85
- GOTO command 73
- `goto` function key 84, 107
- GOTO statement 181
- GotoXY procedure 210, 260
- Graphics 211
  - buffer 213

H

- Halt procedure 207, 230
- Handshake, RS-232 41, 112
- Hard case 111
- Heading
  - function 151
  - procedure 151
  - program 127
  - subroutine 151
- Heap
  - dynamic memory 189
  - memory 51
  - memory area 317
- heapError variable 193, 217
- Hexadecimal number 6, 123
- Hi function 230
- Hide block 81

I

- I compiler directive 271
- I compiler flag 268
- IBM character set 49
- Identifier
  - case of 122
  - conditional compilation 269
  - constant 133
  - EBNF 119
  - global 129
  - global prefixed 155
  - label 131
  - local 155
  - Pascal 121
  - type 135
- Identity in type 145
- IF command 73
- IF statement 176
- IFDEF compiler option 270
- IFEXIST command 73
- IFNDEF compiler option 270
- IFOPT compiler option 271
- Implementation unit 27, 185-186
- Import
  - circular 187
  - unit 187
- IN key 10
- Inc procedure 231
- Include file 271
- Indentation 121
  - automatic 87
- Information, debug 15, 99, 268
- Initialize disk 67
- Initialized variable 148
- INLINE statement 182
  - CPU instruction 321
- Input
  - line edit 36
  - MEMO mode 89
- input variable 199, 217
- INS key 9
- Insert
  - file in editor 84
  - floppy disk 115
- `insert` function key 84
- Insert procedure 231

InsLine procedure 260
Instruction pointer 103
Instruction, CPU 321
Int function 232
  CAL mode 60
Integer
  number 123
  operator 161, 166
  overflow 161
  type 136
Integer type 215
Interface 110
  cassette 114
  connect PB-2000 111
  disk 114
  parallel 113
  remove PB-2000 112
  serial 112
Interface unit 27, 185
Internal variable format 273
Interpreter 3
IOResult function 195, 232
ISO character set 297

K

Key
  ANS 8
  beeper 52, 62
  BRK 10, 294
  BS 9
  CAL 9
  CALC 10
  CAPS 8
  CLS 9
  code 291
  contrast 11
  cursor 9
  dead 291
  DEL 9
  double-code 291
  EXE 10, 13
  EXE in MENU 45
  extended code 291
  function 8
  IN 10
  INS 9
  MEMO 10
Key *(continued)*
  MENU 9
  OUT 10
  power-on code 295
  programmable 35, 50, 294
  shift 8
  space bar 10
  TAB 11
  umlaut 291
Key function
  CAL mode 60
KeyPressed function 260
keyStatus variable 256
Keyword 122
Kill
  See Delete
`kill` function key 47, 54

L

Label 175
  batch file 71
  declaration 131
Language
  syntax 119
Length function 232
Level
  main 129
  subroutine 155
Line
  edit 36
  goto number 85
  wrap-around in editor 87
`line` function key 85, 108
Line procedure 212, 261
Linking, dynamic 185
List files 66
Ln function 232
  CAL mode 60
Lo function 233
Load file 47
  disk 54
`load` function key 47, 54
Local
  identifier 155
  variable, debugger 104
Log function 233
  CAL mode 60

Long pointer 138
LongInt type 215
LongPtr type 215
LPT: device 38

M

M compiler directive 271
Main battery 285
Main level 129
Main part 129
  subroutine 155
Mark
  block 80
  MENU file 43
Mark procedure 190, 233
Mathematic functions
  CAL mode 59
Max function 233
MaxAvail function 192, 234
MD-100 interface 110
MemAvail function 192, 234
MEMO key 10
MEMO mode 89
  function memory 95
  input 89
  search 90
Memory 51
  compiler setting 271
  dynamic 189
  function calculation 93
  heap 317
  numeric variable 318
  stack 317
  system 318
  variable 148
`memory` function key 51
Menu
  dimmed 8
  function 8
  line in editor 87
MENU key 9
MENU mode 43
  disk 46, 53
Message, error 303
Min function 234
Mode
  CAL 55
  MEMO 89
  MENU 43
Move block 82
`move` function key 82
Move procedure 235
Multi-dimensional array 142

N

`name` function key 46, 54
Name, file 37-38
Near pointer 138
`new ad` function key 106
NEW ALL button 11
`new ed` function key 45
`new ip` function key 103
New procedure 190, 235
`next` function key 79
NextKey function 261
NIL constant 167
Notations in manual 5
Null character 291
Number 123
Numeric variable 58
  area 318

O

Odd function 236
Ofs function 236
OM-51P Prolog 109
OM-53B BASIC 109
`on/off` function key 81
Open
  file 195
  user text type 202
Open array parameter 154
Operator
  address 168
  arithmetic 160, 166
  binary 160
  boolean 162, 167
  character 164
  enumeration 162
  integer 161, 166
  integer overflow 161
  pointer 166

Operator *(continued)*
  priority 159
  real 166
  real and integer 162
  reference 169
  set 165
  string 164
  unary 166
Option, compiler 267
Ord function 236
Ordinal
  type 136
  type format 273
  value 137
OUT key 10
output variable 199, 217
Output window, debugger . . . 100
Overflow
  integer operator 161
  real number 138
  real test 268
  stack test 269

P

Page procedure 237
Parallel interface 113
ParamCount function 237
Parameter
  batch file 71
  command line 61
  internal format 276
  open array 154
  structured value 152
  subroutine 152
  untyped 153
  value 152
  variable 153
  variable string test 269
ParamStr function 238
Parity, RS-232 40
.PAS extension 38
Pascal 4
  standard 279
PAUSE command 74
PB-2000 7
Pi function 238
  CAL mode 60
Pixel 211
  draw line 212
  read 212
  set 211
Plot procedure 211, 262
Point function 212, 262
Pointer
  base type 138
  dereference 171
  NIL test 269
  operator 166
  system 139
  type 138
  type format 274
Pointer type 215
Pos function 238
Power
  auto off time 52
  key code on 295
  PB-2000 285
  resume at on 14
  switch PB-2000 7
Power function 239
PowerInt function 239
Precautions
  PB-2000 12
  RAM pack 109
  Release procedure 192
Pred function 239
Prefixed global identifier 155
PRESET command 67
Preset file 24, 53, 67
`preset` function key 53
`prev` function key 85, 108
Previous cursor position 85
Print file 50
`print` function key 50
Printer 114
  file name 39
Priority of operators 159
PRN: device 38
Procedure
  body 155
  call 183
  declaration 151
  forward 156
  heading 151

Procedure *(continued)*
  level 155
  main part 155
  parameter 152
  type 140
  type format 274
`prog.k` function key 50
Program
  body 129
  debug 65
  execute 45, 68
  heading 127
  Pascal 127
  terminate 207
Programmable character 300
Programmable key . 35, 50, 65, 294
Prolog 109
Protocol, communication 39
Ptr function 240

Q

QuickPascal 280

R

R compiler flag 269
RAM pack 109
  insert 109
Random function 240
Randomize procedure 241
Range error test 269
`raw` function key 105
Raw window, debugger 105
Read
  pixel 212
  user text type 202
Read procedure
  text file 241
  typed file 241
ReadKey function 262
ReadLn procedure 241
Real
  number 123
  operator 162, 166
  overflow test 268
  type 138
  type format 274
Real type 215
RealError function 242
Record
  dereference 171
  field 140
  initialized variable 148
  tag type 141
  type 140
  type format 274
  variant part 140
Redirection, file 61
Register, save 277
Release procedure 190, 242
  precautions 192
Remove floppy disk 115
REN command 68
Rename file 46, 68, 195
  disk 54
Rename procedure 243
Repeat search/replace 79
REPEAT statement 178
`replac` function key 78
Replace
  battery PB-2000 285
  disk 54, 115
  editor 78
  flags, editor 79
  global, editor 80
  repeat, editor 79
  ROM card 289
  without ask, editor 80
Reserved word 122
RESET button 11
Reset procedure 243
Resume at power on 14
Return type, function 151
Rewrite procedure 244
Rnd function
  CAL mode 60
ROM card 109, 289
Round function 244
RoundDigit function 245
RoundTo function 244
  CAL mode 60
RP-33 RAM pack 109

RS-232
  cable 113
  file name 39
  handshake 112
  interface 112
Run
  See also Execute
  debug program 45, 65
  program 45, 68
RUN command 68
Run-time error 303
RunError procedure . . . 207, 245

S

S compiler flag 269
S file type 39
Save
  block 83
  file 48
  register 277
`save` function key 48, 83
Screen 7
  See also Display
Search 78
  flags 79
  MEMO 90
  repeat 79
  replace 78
  without case 80
`search` function key . 77-78, 107
SECOND.PAS 19
Seek procedure 246
SeekEof function 246
SeekEoln function 246
Seg function 247
Segment, memory 273
Select
  MENU file 43
  MENU file group 44
Separate compilation 27
Serial interface 112
  cable 113
Set
  base type 143
  empty 167
  operator 165
  pixel 211

Set *(continued)*
  type 143
  type format 275
  value 167
`set bp` function key 102
`set` function key 52
SetTextBuf procedure 247
`setup` function key 86, 108
Shift key 8
Short pointer 138
Short-circuit Boolean 163
ShortInt type 215
SI/SO control, RS-232 41
Sin function 247
  CAL mode 59
Single-character symbol 124
Single-step, debug 101
SinH function 248
  CAL mode 59
SizeOf function 248
Sorting 155
Sound procedure 263
Source file 15
Space bar key 10
Special character, EBNF 119
Specifier, file 37
Speed
  cassette interface 114
  execution 99
  RS-232 40
Sqr function 248
Sqrt function 249
  CAL mode 60
Square bracket, EBNF 120
Stack
  memory 51
  memory area 317
  overflow test 269
Standard Pascal 279
Start DL-Pascal 12
Start ROM card 290
Statement 175
  compound 175
Step
  See Execute
`step` function key 100
Stop bits, RS-232 40

Str procedure 249
String 124
  dereference 170
  operator 164
  parameter type test 269
  type 139
  type format 274
Structured
  type 135
  value parameter 152
Subexpression 167
Subrange
  base type 137
  type 137
Subroutine
  body 155
  declaration 151
  forward 156
  heading 151
  level 155
  main part 155
  parameter 152
  reference 169
  type 140
  type format 274
Succ function 250
Swap function 250
Switch, power PB-2000 7
Symbol
  double-character 125
  Pascal 121
  single-character 124
Syntax of language 119
System
  memory 318
  pointer 139
  unit 128, 215

T

T compiler flag 269
T file type 39
TAB key 11
Tabulator interval 52
Tag type, record 141
Tan function 251
  CAL mode 59
TanH function 251
  CAL mode 59
Terminate
  execution 207
  user procedure 208
Text
  batch file 72
  file 198
  file type 144
  type format 275
  user type 200
Text type 215
textAttr variable 257
TextDisplay procedure 263
TimerOut function 263
TimerStart procedure 264
TO keyword 179
Trigonometric functions
  CAL mode 59
True value 137, 215
Trunc function 251
Turbo Pascal 4, 280
Type
  alias 135
  array 142
  casting 169
  character 136
  compatibility 144
  conversion 169
  declaration 135
  enumeration 137
  file 143
  function 140
  identity compatibility 145
  integer 136
  ordinal 136
  pointer 138
  procedure 140
  real 138
  record 140
  set 143
  string 139
  structured 135
  subrange 137
  subroutine 140
  user text 200

Type of file 39
Typed file 144, 196

U

U file type 39
Umlaut key 291
Unary operator 166
UNDEF compiler option 270
Underflow, real number 138
undo function key 81
Unhide block 81
.UNI extension 38
Unit 185
  checksum 27
  Crt 255
  file name 38
  implementation 27, 186
  import uses 187
  interface 27, 185
  System 128, 215
  uses declaration 128
UNTIL keyword 178
Untyped
  file 144, 197
  parameter 153
Upcase function 251
UpdateRegion procedure 213, 264
User features, editor 86
User terminate procedure 208
User text type 200
Uses
  circular import 187
  declaration 128
  unit 187

V

V compiler flag 269
Val procedure 252
Value
  argument 167
  parameter 152
  set 167
  type conversion 169
Variable
  absolute 147
  CAL mode 58
  debug 104
  declaration 147
  dereference 170
  dynamic 189
  enter data, debugger 105
  FOR control 179
  function memory 93
  global 147
  initialized 148
  internal format 273
  local/global 104
  memory in CAL 51
  numeric area, CAL 318
  parameter 153
  reference 169
  type conversion 169
  WITH priority 181
Variant part of record 140
VERIFY command 69
Verify file 69
Vertical bar, EBNF 119

W

WhereX function 210, 264
WhereY function 210, 265
WHILE statement 178
Wild card 37, 44
WindMaxX function 265
WindMaxY function 265
WindMinX function 265
WindMinY function 265
Window 209
  code, debugger 100
  cursor 210
  data, debugger 104
  output, debugger 100
  raw, debugger 105
  set 209
Window procedure 209, 266
WITH statement 180
Word type 215
Wrap-around, editor 87

Write
  batch file 72
  user text type 203
Write procedure
  text file 253
  typed file 252
WriteLn procedure 253

X

xchg function key 104
XON/XOFF handshaking . . . 41